SQL and NoSQL Course

Mohammed Abdul Sami
Database Consultant & Trainer

About This Document

Course material for the SQL and NoSQL course designed by Mohammed Abdul Sami. This material is intended for the use of anyone with interest to learn Databases.

This course is primarily focused on teaching RDBMS concepts, SQL with MySQL and NoSQL concepts and NoSQL practice with MongoDB

About Trainer

Mohammed Abdul Sami is an experienced System Administrator & DBA having more than 20 years of experience in managing Database Servers, Linux and Unix servers.

Since 2014 he is mostly involved in trainings and a frequent instructor at Oracle University for MySQL and Solaris Courses.

For internal reference only not for distribution or sale

Contents

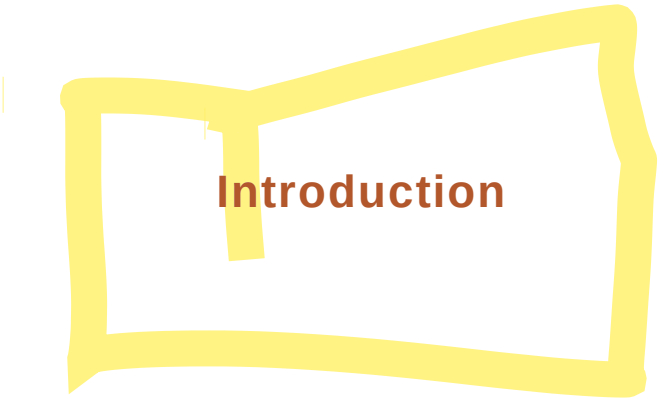
by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

RDBMS

(SQL Databases)

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale



For internal reference only not for distribution or sale

RDBMS

RDBMS stands for Relational Database Management System. RDBMS recommends data to be logically organized in Relations (tables), Tuples (rows or records), Attributes (fields or columns). Robustness and power of RDBMS model derives from the use of simple tabular structure to represent the data. Data is represented uniformly in a tabular structure as it is easy to organize and manipulate it.

Most of the RDBMS theory is based on mathematical set theory. It was first proposed by Dr. Edgar Frank Codd, A computer scientists working at IBM company in 1970 and he improvised the theory throughout 1970s & 1980s. Theoretical development of the model continues till this day.

The Relations (tables) in RDBMS theory corresponds to mathematical sets, tuples (rows) correspond to members of set. Many of the principles from mathematical set theory is applied in RDBMS.

RDBMS Terminology

Relation (Table)

In Relational database model, a table is a collection of data elements organised in terms of rows and columns. A table is also considered as a convenient representation of relations. But a table can have duplicate row of data while a true relation cannot have duplicate data.

Tuple (Record or Row)

A single entry in a table is called a Tuple or Record or Row. A tuple in a table represents a set of related data.

Attribute (Field or Column name)

A table consists of several records(row), each record can be broken down into several smaller parts of data known as Attributes.

Domain (Datatype)

When an attribute is defined in a relation(table), it is defined to hold only a certain type of values, which is known as Attribute Domain.

Keys

A relation key is an attribute which can uniquely identify a particular tuple(row) in a relation(table). It is also called as Primary Key

For internal reference only not for distribution or sale

Foreign Key

An attribute which can link two relations. For example, a course-id uniquely identifies all the offered courses along with their details in Course_table and which student has opted for which course in the student-table.

Degree

The degree of a relation/table is the number of columns it contains.

Cardinality

The cardinality of a relation/table is the number of rows it contains.

Instance

The data stored in database at a particular moment of time is called instance of database. .

Schema

A relation schema describes the structure of the relation, with the name of the relation(name of table), its attributes and their names and datatype.

E. F. Codd's Rules for RDBMS

Dr. Edgar Frank Codd was an English computer scientist who, while working for IBM, invented the relational model for database management, the theoretical basis for relational databases.

Rule 1 – Information Rule

All information is explicitly and logically represented in exactly one way – by data values in tables.

If an item of data does not reside somewhere in a table in the database, then it does not exist.

Rule 2 – Rule of Guaranteed Access

Every item of data must be logically addressable by resorting to a combination of table name, primary key value and a column name.

Rule 3 – The Systematic Treatment of Null Values

The RDBMS handles records that have unknown or inapplicable values in a pre-defined fashion.

The RDBMS distinguishes between zeros, blanks and nulls in the records hand handles such values in a consistent manner that produces correct answers, comparisons and calculations.

For internal reference only not for distribution or sale

Through the set of rules for handling nulls, users can distinguish results of the queries that involve nulls, zeros and blanks.

Rule 4 – The Database Description Rule

The description of a database and in its contents are database tables and therefore can be queried on-line via the data manipulation language.

There must be a Data Dictionary within the RDBMS that is constructed of tables and/or views that can be examined using SQL.

Rule 5 – The Comprehensive Sub-language rule

A RDBMS may support several languages. But at least one of them should allow user to do all of the following:

- Define tables and views
- Query and update the data
- Set integrity constraints
- Set authorizations
- Define transactions.

RDBMS must be completely manageable through its own dialect of SQL.

Rule 6 – The View Updating Rule

All views that can be updated in theory, can also be updated by the system.

Data consistency is ensured since the changes made in the view are transmitted to the base table and vice-versa.

Rule 7 – High Level Insert, Update and Delete Rule

The RDBMS supports insertions, updating and deletion at a table level.

This means that data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables. This rule states that insert, update, and delete operations should be supported for any retrievable set rather than just for a single row in a single table.

Rule 8 – The Physical Independence Rule

For internal reference only not for distribution or sale

The execution of adhoc requests and application programs is not affected by changes in the physical data access and storage methods.

Database administrators can make changes to the physical access and storage method which improve performance and do not require changes in the application programs or requests. Here the user specified what he wants and need not worry about how the data is obtained.

Rule 9 – Logical Data Interdependence Rule

Logical changes in tables and views such adding/deleting columns or changing fields lengths need not necessitate modifications in the programs or in the format of adhoc requests.

The database can change and grow to reflect changes in reality without requiring the user intervention or changes in the applications. For example, adding attribute or column to the base table should not disrupt the programs or the interactive command that have no use for the new attribute.

Rule 10 – Integrity Independence Rule

Integrity constraints must be specified separately from application programs and stored in the catalog. It must be possible to change such constraints as and when appropriate without unnecessarily affecting existing applications.

The following Integrity rules(Constraints) should apply to every RDBMS :-

No component of a primary key can have missing values – this is the basic rule of Entity Integrity.

For each distinct foreign key value there must exist a matching primary key value in the same domain. Conformation to this rule ensures what is called Referential integrity.

Rule 11 – Distribution Rule

Application programs and adhoc requests are not affected by change in the distribution of physical data.

Improved systems reliability since application programs will work even if the programs and data are moved in different sites.

Rule 12 – No Subversion Rule

If the RDBMS has a language that accesses the information of a record at a time, this language should not be used to bypass the integrity constraints.

For internal reference only not for distribution or sale

Normalization

Database Anomalies

Anomalies in Database design are faults in organizing data which may lead to in efficient way of managing database.

Update anomalies

If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.

Deletion anomalies

We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.

Insert anomalies

We tried to insert data in a record that does not exist at all.

Normalization

Normalization is a method to remove all these anomalies and bring the database to a consistent state.

The primary objective of the normalization is that the data is structured in a way that all the attributes are grouped along with primary key which ensures unique identification of all tuples. To achieve this we may have to decompose the data into many relations.

To achieve consistent database the below Normalization forms available

Un-Normalized Form (UNF)

If a table contains non-atomic values at each row, it is said to be in UNF. An atomic value is something that can not be further decomposed

Students Data

Name	Address	Phone No	Course	Duration	Course Fee	Fee Paid	Final Marks	Division	Instructor
Niel	10, Rd-5	875642,987543	MySQL	40	25000	20000	97	Distinction	Sami

For internal reference only not for distribution or sale

Students Data

Name	Address	Phone No	Course	Duration	Course Fee	Fee Paid	Final Marks	Division	Instructor
Dave	city center	874321	Oracle	80	50000	50000			Aaron
Mia	2nd avenue	982133	Postgresql	32	30000	30000	89	First	Sami
Riley	barrows	982156	Oracle	80	50000	45000	73	Second	Aaron
Emma	5, Rd-8	870987, 870543	MySQL	40	25000	25000	75	Second	Sami
George	City Center		MySQL	40	25000	25000	91	First	Sami
Jane	2nd Avenue	984321	Postgresql	32	30000	30000	93	First	Sami

First Normal Form (1NF)

A relation is said to be in 1NF if every attribute contains no non-atomic values and each row can provide a unique combination of values.

Students Data- 1st NF

Name	Address	Phone No	Home Phone	Course	Duration	Course Fee	Fee Paid	Final Marks	Division	Instructor
Niel	10, Rd-5	875642	987543	MySQL	40	25000	20000	97	Distinction	Sami
Dave	city center	874321		Oracle	80	50000	50000			Aaron
Mia	2nd avenue	982133		Postgresql	32	30000	30000	89	First	Sami
Riley	barrows	982156		Oracle	80	50000	45000	73	Second	Aaron
Emma	5, Rd-8	870987	870543	MySQL	40	25000	25000	75	Second	Sami
George	City Center			MySQL	40	25000	25000	91	First	Sami
Jane	2nd Avenue	984321		Postgresql	32	30000	30000	93	First	Sami

For internal reference only not for distribution or sale

Second Normal Form (2NF)

A relation is said to be in 2NF if it is already in 1NF and each and every attribute fully depends on the primary key of the relation.

Course Data

Course	Duration	Course Fee
Oracle	80	50000
MySQL	40	25000
Postgresql	32	30000

Students Data in 2nd NF

Name	Address	Phone No	Home Phone	Course	Fee Paid	Final Marks	Division	Instructor
Niel	10, Rd-5	875642	987543	MySQL	20000	97	Distinction	Sami
Dave	city center	874321		Oracle	50000			Aaron
Mia	2nd avenue	982133		Postgresql	30000	89	First	Sami
Riley	barrows	982156		Oracle	45000	73	Second	Aaron
Emma	5, Rd-8	870987	870543	MySQL	25000	75	Second	Sami
George	City Center			MySQL	25000	91	First	Sami
Jane	2nd Avenue	984321		Postgresql	30000	93	First	Sami

Third Normal Form (3NF)

A relation is said to be in 3NF, if it is already in 2NF and there exists no transitive dependency in that relation.

For internal reference only not for distribution or sale

Course Data- 3rd NF

CourseID	Course	Duration	Course Fee
100	Oracle	80	50000
200	MySQL	40	25000
300	Postgresql	32	30000

Students Data- 3rd NF

StudentID	Name	Address	Phone No	Home Phone	CourseID	Fee Paid	Final Marks	Instructor
1551	Niel	10, Rd-5	875642	987543	MySQL	20000	97	Sami
1552	Dave	city center	874321		Oracle	50000		Aaron
1553	Mia	2nd avenue	982133		Postgresql	30000	89	Sami
1554	Riley	barrows	982156		Oracle	45000	73	Aaron
1555	Emma	5, Rd-8	870987	870543	MySQL	25000	75	Sami
1556	George	City Center			MySQL	25000	91	Sami
1557	Jane	2nd Avenue	984321		Postgresql	30000	93	Sami

Divisions Chart - 3rd NF

Division	Min Marks	Max Marks
Distinction	95	100
First	85	94
Second	75	84
Passed	65	74

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Divisions Chart - 3rd NF

Division	Min Marks	Max Marks
Need Improvement	0	64

Fourth Normal Form (4NF)

When attributes in a relation have multi-valued dependency, further Normalization to 4NF.

Course Data - 4th NF

CourseID	Course	Duration	Course Fee
100	Oracle	80	50000
200	MySQL	40	25000
300	Postgresql	32	30000

Students Data - 4th NF

StudentID	Name	Address	Phone No	Home Phone	CourseID	Fee Paid	Final Marks
1551	Niel	10, Rd-5	875642	987543	MySQL	20000	97
1552	Dave	city center	874321		Oracle	50000	
1553	Mia	2nd avenue	982133		Postgresql	30000	89
1554	Riley	barrows	982156		Oracle	45000	73
1555	Emma	5, Rd-8	870987	870543	MySQL	25000	75
1556	George	City Center			MySQL	25000	91
1557	Jane	2nd Avenue	984321		Postgresql	30000	93

For internal reference only not for distribution or sale

Instructor 4th NF

InstructorID	Name	address
DB01	Sami	Arlington Drive
DB02	Aaron	4th Avenue

Divisions Chart - 4th NF

Division	Min Marks	Max Marks
Distinction	95	100
First	85	94
Second	75	84
Passed	65	74
Need Improvement	0	64

Instructor / Student Map - 4th NF

InstructorID	StudentID
DB02	1551
DB01	1552
DB02	1553
DB01	1554
DB02	1555
DB02	1556
DB02	1557

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Workbook

Exercise - 1

Normalize the below health history data upto 3rd NF

HEALTH HISTORY REPORT

PET ID	PET NAME	PET TYPE	PET AGE	OWNER	VISIT DATE	PROCEDURE
246	ROVER	DOG	12	SAM COOK	JAN 13/2002	01 - RABIES VACCINATION
					MAR 27/2002	10 - EXAMINE and TREAT WOUND
					APR 02/2002	05 - HEART WORM TEST
298	SPOT	DOG	2	TERRY KIM	JAN 21/2002	08 - TETANUS VACCINATION
					MAR 10/2002	05 - HEART WORM TEST
341	MORRIS	CAT	4	SAM COOK	JAN 23/2001	01 - RABIES VACCINATION
					JAN 13/2002	01 - RABIES VACCINATION
519	TWEEDY	BIRD	2	TERRY KIM	APR 30/2002	20 - ANNUAL CHECK UP
					APR 30/2002	12 - EYE WASH

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Exercise - 2

Normalize the below Invoice data unto 3rd Normal Form

INVOICE

HILLTOP ANIMAL HOSPITAL DATE: JAN 13/2002

INVOICE # 987

MR. RICHARD COOK

123 THIS STREET

MY CITY, ONTARIO

Z5Z 6G6

PET	PROCEDURE	AMOUNT
ROVER	RABIES VACCINATION	30.00
MORRIS	RABIES VACCINATION	24.00
	TOTAL	54.00
	TAX (8%)	4.32
	AMOUNT OWING	58.32

For internal reference only not for distribution or sale

Introduction to MySQL

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Introduction to MySQL

The MySQL is a very fast, multithreaded, multi-user, and robust RDBMS server. MySQL Server is intended for mission-critical, heavy-load production systems as well as for embedding into mass-deployed software.

MySQL Database is dual licensed.

1. Open Source under the terms of the GNU General Public License
2. Commercial license from Oracle

History

- In 1995 Michael Widenius (Monty), David Axmark and Allan Larsson founded MySQLAB in Sweden. First version of MySQL appeared in 23 May, 1995
- Windows version was released on 8 January 1998 for Windows 95 and NT
- MYSQL goes Open Source and releases software under the terms of the GPL(General Public License) in year 2000
- Sun Microsystems acquired MySQL AB in 2008
- Oracle acquired Sun Microsystems on 27 January 2010

Branches & Forks

MariaDB

In 2008 Sun Microsystems bought MySQL, Sun being itself later acquired by Oracle, in 2010. After the acquisition, the development process has changed. The team has started to release new MySQL versions less frequently, so the new code is less tested. There were also less contributions from the community.

In 2009 Monty Widenius, the founder of MySQL, left the company and created a new one, called The Monty Program. He started a new fork called MariaDB

Drizzle

In 2008 Brian Aker, chief architect of MySQL, left the project to start a new fork called Drizzle. While Oracle initially funded the project, Drizzle is now funded by Rackspace.

Percona Server

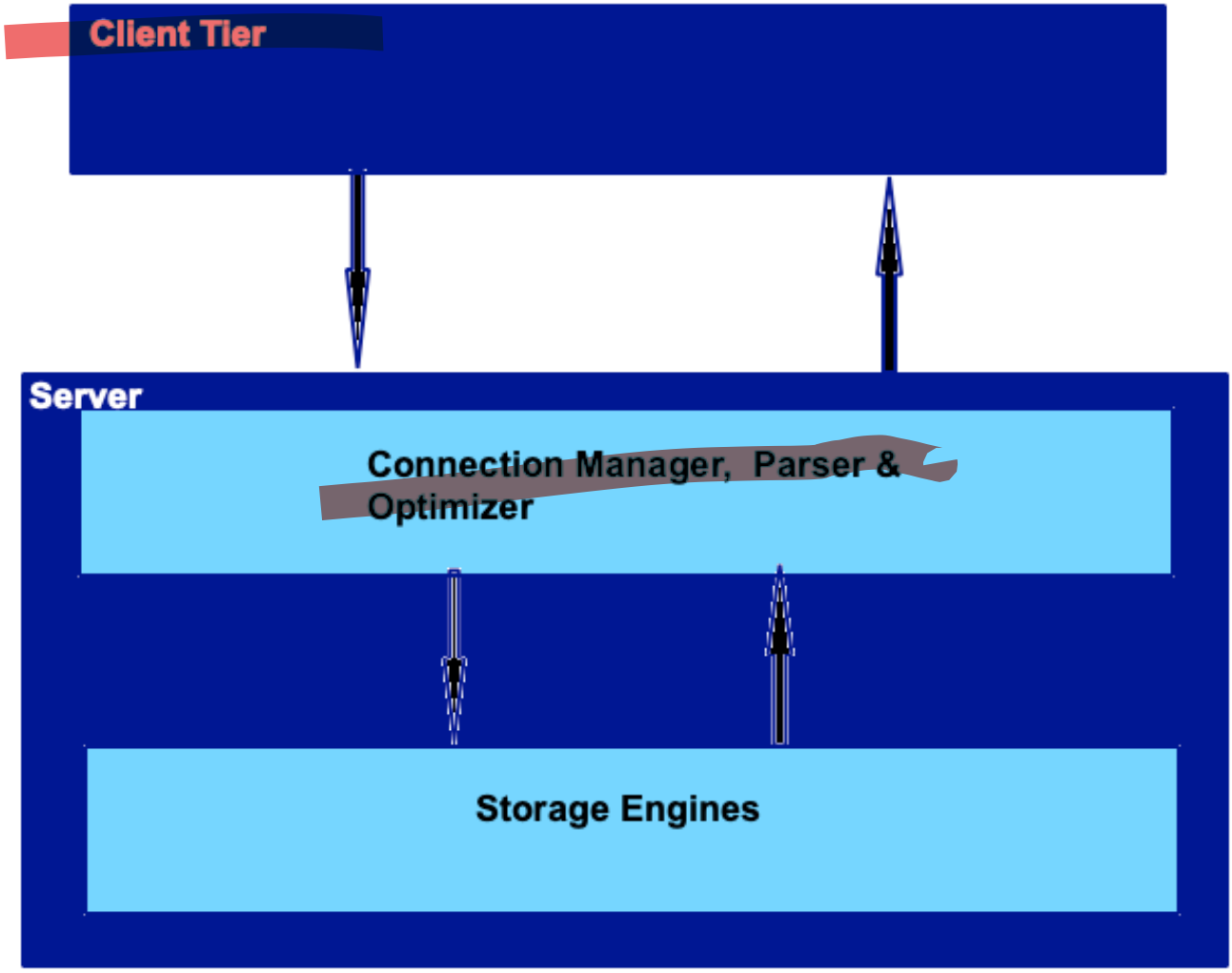
Percona Server is a MySQL fork maintained by Percona. It provides the ExtraDB Storage Engine, which is a fork of InnoDB, and some patches which mainly improve the performance.

For internal reference only not for distribution or sale

Our Delta

Our Delta is another fork, maintained by Open Query. The first branch, which has number 5.0, is based on MySQL 5.0. The 5.1 branch is based on MariaDB. OurDelta includes some patches developed by the community or by third parties. OurDelta provides packages for some GNU/Linux distributions: Debian, Ubuntu, Red Hat/CentOS. It is not available for other systems, but the source code is freely available.

Client / Server Model



MySQL Components

Connection Manager

Connection manager threads handle client connection requests on the network interfaces that the server listens to, Unix / Linux sockets, Windows Shared Memory connections and windows named

For internal reference only not for distribution or sale

pipes. User credentials are validated before establishing connection. Connection manager threads associate each client connection with a thread dedicated to it that handles authentication and request processing for that connection.

A new thread is created when necessary but avoids doing so by consulting the thread cache first to see whether it contains a thread that can be reused for the connection.

When a connection ends, its thread is returned to the thread cache.

SQL Interface

SQL interface provides mechanism to receive commands and transmit results back to user. It is designed to accept ANSI standard SQL queries but some of queries may use MySQL specific syntax.

Parser

When a client issues a query, a new thread is created and the SQL statement is forwarded to the parser for syntactic validation (or rejection due to errors).

Optimizer

These strategies may include rewrite request, make sure to read the order of thetable, decide which indexes

Caches & Buffers

Caches & Buffers subsystem is responsible for ensuring most frequently used data is available in most efficient manner. Caches dramatically increase the response time for such data as that is available in memory eliminating the need to read from disk.

Storage Engines

This is one of the distinct feature of MySQL which allows to tune database performance by selecting suitable storage engine for their application.

Storage engines responsible for the physical storage and retrieval of data from disk files.

Clients

MySQL Supports any client which can request data using SQL language. Officials clients are

- mysql (default command line client)
- MySQL Workbench (GUI Client)

For internal reference only not for distribution or sale

Storage Engines

A storage engine is a software module that a database management system uses to create, read, update data from a database. There are two types of storage engines in MySQL. Transactional and non-transactional.

Transactional Storage Engine: A MySQL storage Engine which supports DML Transactions.

Non-Transactional Storage Engine: A MySQL storage Engine which presents data as is and does not support tractions.

MySQL supports several storage engines that act as handlers for different table types. MySQL storage engines include both those that handle transaction-safe tables and those that handle non-transaction-safe tables.

Some of the supported storage engines

- MyISAM
- InnoDB
- Memory
- CSV
- Merge
- Archive
- Federated
- Blackhole
- Example

For internal reference only not for distribution or sale

Structured Query Language

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

SQL (Structured Query Language)

SQL was initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce for manipulating, retrieving and storing data from RDBMS in the early 1970s.This version, initially called SEQUEL (Structured English Query Language)

Also parallelly developed at Oracle Corporation and Ingres Database. Later it has been standardize under ANSI SQL standards.

Sub Languages

Further SQL language has been classified in the following sub-languages

DDL

Data Definition Language. Useful create design relational schemas and create and manage other objects. It consists of 4 statements

CREATE

ALTER

DROP

TRUNCATE

DML

Data Manipulation Language. used to save, retrieve and manipulate data. It consists of 4 statements

INSERT

UPDATE

DELETE

SELECT

TCL

Transaction Control Language. In a multiuser system transactions ensure concurrent DML operations happen without any conflicts and ensures data consistency. It consists of 5 statements

BEGIN or START TRANSACTION

COMMIT

ROLLBACK

For internal reference only not for distribution or sale

SAVEPOINT

ROLLBACK TO SAVEPOINT

DCL

Data Control Language. It helps In implementing access restriction to the data so that data can only be accessed or manipulated by authorized user. It consists of 2 statements

GRANT

REVOKE

Data Types

Data type defines what type data will be stored in a particular column.

- Numeric Data Types
- Date and Time Data Types
- String Data Types
- Special Data Types

Numeric Datatypes

Integer Types

below are Integer datatypes. Integer datatypes don't accept precision value

Name	Storage Size
tinyint	1
smallint	2
mediumint	3
int	4
bigint	8

For internal reference only not for distribution or sale

Fixed Numeric

Fixed numeric datatypes retains exact same value which is inputted without rounding off.

Name	description
decimal (m,d)	M is total number of digits and d precision

Approximate Numeric

Approximate numeric datatypes are similar to decimal but values may be changed due roundoff.

Name	description
float (m,d)	M is total number of digits and d precision
double (m, d)	M is total number of digits and d precision

bit datatypes

bit datatype allows to store bit values (0,1)

Name	description
bit (N)	N is the number of bit values which can be stored.

Date and Time types

Date and Time type allows to save date time data.

Name	Format
DATE	'YYYY-MM-DD'

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Name	Format
TIME	'hh:mi:ss'
DATETIME	'YYYY-MM-DD hh:mi:ss'
TIMESTAMP	'YYYY-MM-DD hh:mi:ss'
YEAR	YYYY'

Character / String types

Allows to store string type data.

Name	description
char	Fixed size character string
varchar	Variable size character string
text	Variable size text
mediumtext	Variable size text
longtext	Can store large amount text information

Unformatted Data

These data types allows to store large sized unformatted data like picture, video clips, sound clips etc

Name
blob
longblob

For internal reference only not for distribution or sale

Name
binary
varbinary

Special Datatypes

Name	description
JSON	Allows valid JSON documents to stored
ENUM	Enumerated type allows user to input the values in strings and saves them as numeric codes in a small storage footprint
SET	Allows values from pre-defined list to stored.

Data Definition Language

DDL consists of 4 statements namely CREATE, ALTER, DROP and TRUNCATE. This sub language is helpful in defining table structure, (like name of table, columns names and their data types) change the definition of table (like adding a new columns) or dropping the table.

Creating a Database

A database is a collection of database objects such as tables etc. It allows us to group the objects for efficient management.

Syntax:

```
CREATE DATABASE  <DATABASENAME>;
```

Example:

```
CREATE DATABASE students;
```

USE command

USE command is helpful to switch the database to access tables from it.

For internal reference only not for distribution or sale

```
USE <database name>;
```

SHOW command

SHOW command helps in listing all the existing databases and tables which are present in the current database.

```
SHOW  DATABASES;
```

```
SHOW TABLES;
```

Creating Tables

- A table is a named collection of rows
- Each table row has same set of columns
- Each column has a data type and optionally a constraint

Tables can be created using the CREATE TABLE statement

Syntax:

```
CREATE TABLE table_name
( [column_name data_type [ column_constraint], ..... );
```

```
Example:
```

```
CREATE TABLE COURSES (courseid  int,  course_name varchar(25),
duration_in_hrs int);
```

DESCRIBE command

Describe command shows the table structure of a already created table/

```
DESCRIBE  <table name>;
```

Changing the table definition (ALTER TABLE)

ALTER TABLE is used to modify the structure of a table

Can be used to:

- Add a new column
- Modify an existing column
- Add a constraint
- Drop a column
- Rename a column, constraint or table

For internal reference only not for distribution or sale

Syntax:

```
ALTER TABLE table_name ADD column_name datatype column_constraint
[FIRST|AFTER <columns>];

ALTER TABLE table_name  DROP COLUMN column_name;

ALTER TABLE table_name  RENAME COLUMN column_name  TO new_column_name;

ALTER TABLE table_name MODIFY column_name <new definition>;

ALTER TABLE table_name ADD CHECK expression;

ALTER TABLE table_name ADD CONSTRAINT constraint_name constraint_definition;

ALTER TABLE table_name RENAME TO new_table_name;
```

DROP TABLE

The DROP TABLE statement is used to drop a table All data and structure is deleted

Related indexes and constraints are dropped Syntax:

```
DROP TABLE [ IF EXISTS ] name [, ...];
```

TRUNCATE TABLE

Truncate table statement deletes all the rows from the given table and empties the table.

```
TRUNCATE  TABLE <TABLE NAME>;
```

Data Manipulation Language

Data Definition Language consists of 4 statements INSERT, UPDTAE, DELETE and SELECT. These statements are useful to add, modify or retrieve data from tables.

INSERT

INSERT statement add a new row to the table.

Synstax:

```
INSERT INTO  <table name>  VALUES  (v1, v2,.....vn);
```

Note: Need to provide values for all the columns in the same order as they are arranged in the table.

Syntax:

INSERT statement can be used to add new row into the table by providing values only for few columns.

Syntax:

For internal reference only not for distribution or sale

```
INSERT INTO <table name>  (column1, column2, ...)   VALUES  (v1, v2,...);
```

SELECT

SELECT statement allows us to retrieve the data from the tables.

Syntax:

```
SELECT   *  FROM  <table name>;
```

Choosing Columns with SELECT:

Syntax:

```
SELECT  column1, column2,...   FROM <table name>;
```

UPDATE

UPDATE statement is useful to change the columns values in the table.

Syntax:

```
UPDATE  <table name>  SET   <column1> = <value>,  <column2> = <value>
[WHERE   <condition>];
```

All the rows matching the giving conditions are changed. If WHERE is omitted then all rows in the table are changed.

DELETE

DALETE statements deletes selected rows or all rows from the table.

Syntax:

```
DELETE  FROM <table name>  [WHERE  <condition>];
```

If WHERE clause is omitted then all rows from the table are deleted.

MySQL Operators

Operators helps to perform various data manipulations like arthimetic operations, comparisons etc.

Arithmetic Operators

Operator	Description
+	Add

For internal reference only not for distribution or sale

Operator	Description
-	Subtract
*	Multiple
/	Divide
%	Modulo

Comparison operators

Operator	
=	Equality
!= or <>	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than Equal to
<=	Less Than and Equal to
BETWEEN	Range comparison
IN (v1, v2,....)	Checks if expression is equal to any one of the given value in list
LIKE	Used for sub string comparison. % represents string os any length & _ represents one character. eg: 'R%' any string starting with R '%R%' Any string which has R in it

For internal reference only not for distribution or sale

Operator	
	'%R' Any string which ends with R
IS NULL	Checks if the given value is null

Logical Operators

Operator	Description
AND	True if Both the given conditions are true
OR	True if Any of the given conditions is true
NOT	Logical negation

Special Operators for Sub Queries

Operator	Description
ALL	True if all sub query results meets the condition
ANY	True if any of the sub query results meets the condition
EXISTS	True if sub query returns results

For internal reference only not for distribution or sale

Constraints

Constraints are used to enforce data integrity

MySQL supports different types of constraints:

- NOT NULL
- CHECK
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY

Constraints can be defined on a table at the time of creation or can also be enforced latter using ALTER TABLE statement.

Primary Key

Any table can have only one primary key. It restricts duplicate values and NULL values in the column or combination of columns. Primary key identifies all rows uniquely within the table it is defined. It is recommended to have a primary key for each table.

Syntax-1

```
CREATE TABLE <table name>  (
  <columns name>   <datatype>   PRIMARY KEY,
<column name2>  <datatype>,
..............
);
```

Syntax-2:

```
CREATE TABLE <table name>  (
  <columns name>   <datatype> ,
<column name2>  <datatype>,
<columns name3>  <datatype>,
PRIMARY KEY (column1, column2......)
);
```

Syntax-3:

```
ALTER TABLE  <table name> ADD  PRIMARY KEY (col1, col2...);
```


For internal reference only not for distribution or sale

Note: Creating a primary key also create a Unique index on the columns.

Unique Key

Like primary key Unique Key also restricts the duplicate values in the columns but it allows NULL values. A table can have more than one Unique Keys

Syntax-1

```
CREATE TABLE <table name>  (
  <columns name>   <datatype>   UNIQUE,
<column name2>  <datatype>,
..............
);
```

Syntax-2:

```
CREATE TABLE <table name>  (
  <columns name>   <datatype> ,
<column name2>  <datatype>,
<columns name3>  <datatype>,
UNIQUE(column1, column2......)
);
```

Syntax-3:

```
ALTER TABLE  <table name> ADD  CONSTRAINT <constraint name> UNIQUE (col1, col2...);
```

Note: Creating a Unique index also enforces UNIQUE constraints.

NOT NULL

NOT NULL constraint disallow NULL values in the columns.

Syntax:

```
CREATE TABLE <table name>  (
  <columns name>   <datatype>   NOT NULL,
<column name2>  <datatype>,
..............
);
```


For internal reference only not for distribution or sale

CHECK Constraint

CHECK constraint ensures that the values entered in the column satisfy the given condition.

Syntax-1

```
CREATE TABLE <table name>  (
  <columns name>   <datatype>,
<column name2>  <datatype> CHECK  (<condition>,
..............
);
```

Syntax-2:

```
CREATE TABLE <table name>  (
  <columns name>   <datatype> ,
<column name2>  <datatype>,
<columns name3>  <datatype>,
CONSTRAINT <constraint name> CHECK (<condition>)
);
```

Syntax-3:

```
ALTER TABLE  <table name> ADD  CONSTRAINT <constraint name> CHECK
(<condition>);
```

FOREIGN KEY

FOREIGN KEY constraint restricts the values in the columns to the available values in the PRIMARY KEY of another table. It establishes the link between two tables by referencing the PRIMARY KEY of another table.

Syntax-1

```
CREATE TABLE <table name>  (
  <columns name>   <datatype>,
<column name2>  <datatype> REFERENCES  <another table name(<column>),
..............
);
```

Syntax-2:

```
CREATE TABLE <table name>  (
```


For internal reference only not for distribution or sale

```
  <columns name>   <datatype> ,
<column name2>  <datatype>,
<columns name3>  <datatype>,
CONSTRAINT <constraint name> FOREIGN KEY (column) REFERENCES
<tablename>(<condition>)
);
```

Syntax-3:

```
ALTER TABLE  <table name> ADD  CONSTRAINT <constraint name> FOREIGN KEY
(<columns>) REFERENCES  <tablename>(col1, col2..);
```

- Constraints can be defined at the column level or table level
- Constraints can be added to an existing table using the ALTER TABLE statement
- Constraints can be declared DEFERRABLE or NOT DEFERRABLE
- Constraints prevent the deletion of a table if there are dependencies

Default values

Any columns can be assigned a default value which will replace NULL in case if the values for the column is not provided.

Syntax-1

```
CREATE TABLE <table name>  (
  <columns name>   <datatype>,
<column name2>  <datatype>  DEFAULT <value>,
..............
);
```

Syntax-2:

```
ALTER TABLE  <table name> ALTER COLUMN <columns> SET DEFAULT <value>;
```

Generated Columns

Value of generated column is computed from the expression given in columns definition.

Example1:

```
CREATE TABLE t1 (
```


For internal reference only not for distribution or sale

```
  first_name VARCHAR(10),
  last_name VARCHAR(10),
  full_name VARCHAR(255) AS (CONCAT(first_name,' ',last_name))
);

SELECT full_name FROM t1;
```

Example1:

```
CREATE TABLE trinagles(
  base  float,
  height float,
  area float AS (base * height * 0.5)
);

SELECT * FROM triangles;
```

Built-In Functions

Numeric Functions

Function	Description
ABS	Returns the absolute value of a number
ACOS	Returns the arc cosine of a number
ASIN	Returns the arc sine of a number
ATAN	Returns the arc tangent of one or two numbers
ATAN2	Returns the arc tangent of two numbers
AVG	Returns the average value of an expression
CEIL	Returns the smallest integer value that is >= to a number

For internal reference only not for distribution or sale

CEILING	Returns the smallest integer value that is >= to a number
COS	Returns the cosine of a number
COT	Returns the cotangent of a number
COUNT	Returns the number of records returned by a select query
DEGREES	Converts a value in radians to degrees
DIV	Used for integer division
EXP	Returns e raised to the power of a specified number
FLOOR	Returns the largest integer value that is <= to a number
GREATEST	Returns the greatest value of the list of arguments
LEAST	Returns the smallest value of the list of arguments
LN	Returns the natural logarithm of a number
LOG	Returns the natural logarithm of a number, or the logarithm of a number to a specified base
LOG10	Returns the natural logarithm of a number to base 10
LOG2	Returns the natural logarithm of a number to base 2
MAX	Returns the maximum value in a set of values
MIN	Returns the minimum value in a set of values
MOD	Returns the remainder of a number divided by another number
PI	Returns the value of PI
POW	Returns the value of a number raised to the power of another number

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

POWER	Returns the value of a number raised to the power of another number
RADIANS	Converts a degree value into radians
RAND	Returns a random number
ROUND	Rounds a number to a specified number of decimal places
SIGN	Returns the sign of a number
SIN	Returns the sine of a number
SQRT	Returns the square root of a number
SUM	Calculates the sum of a set of values
TAN	Returns the tangent of a number
TRUNCATE	Truncates a number to the specified number of decimal places

String Functions

Function	Description
ASCII	Returns the ASCII value for the specific character
CHAR_LENGTH	Returns the length of a string (in characters)
CHARACTER_LENGTH	Returns the length of a string (in characters)
CONCAT	Adds two or more expressions together
CONCAT_WS	Adds two or more expressions together with a separator
FIELD	Returns the index position of a value in a list of values
FIND_IN_SET	Returns the position of a string within a list of strings

For internal reference only not for distribution or sale

FORMAT	Formats a number to a format like "#,###,###.##", rounded to a specified number of decimal places
INSERT	Inserts a string within a string at the specified position and for a certain number of characters
INSTR	Returns the position of the first occurrence of a string in another string
LCASE	Converts a string to lower-case
LEFT	Extracts a number of characters from a string (starting from left)
LENGTH	Returns the length of a string (in bytes)
LOCATE	Returns the position of the first occurrence of a substring in a string
LOWER	Converts a string to lower-case
LPAD	Left-pads a string with another string, to a certain length
LTRIM	Removes leading spaces from a string
MID	Extracts a substring from a string (starting at any position)
POSITION	Returns the position of the first occurrence of a substring in a string
REPEAT	Repeats a string as many times as specified
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)
RPAD	Right-pads a string with another string, to a certain length
RTRIM	Removes trailing spaces from a string

For internal reference only not for distribution or sale

SPACE	Returns a string of the specified number of space characters
STRCMP	Compares two strings
SUBSTR	Extracts a substring from a string (starting at any position)
SUBSTRING	Extracts a substring from a string (starting at any position)
SUBSTRING_INDEX	Returns a substring of a string before a specified number of delimiter occurs
TRIM	Removes leading and trailing spaces from a string
UCASE	Converts a string to upper-case
UPPER	Converts a string to upper-case

Date Functions

Function	Description
CURDATE	Returns the current date.
DATEDIFF	Calculates the number of days between two DATE values.
DAY	Gets the day of the month of a specified date.
DATE_ADD	Adds a time value to date value.
DATE_SUB	Subtracts a time value from a date value.

For internal reference only not for distribution or sale

DATE_FORMAT	Formats a date value based on a specified date format.
DAYNAME	Gets the name of a weekday for a specified date.
DAYOFWEEK	Returns the weekday index for a date.
EXTRACT	Extracts a part of a date.
LAST_DAY	Returns the last day of the month of a specified date
NOW	Returns the current date and time at which the statement executed.
MONTH	Returns an integer that represents a month of a specified date.
STR_TO_DATE	Converts a string into a date and time value based on a specified format.
SYSDATE	Returns the current date.
TIMEDIFF	Calculates the difference between two TIME or DATETIME values.
TIMESTAMPDIFF	Calculates the difference between two DATE or DATETIME values.
WEEK	Returns a week number of a date.
WEEKDAY	Returns a weekday index for a date.
YEAR	Return the year for a specified date

For internal reference only not for distribution or sale

Aggregate Functions

Aggregate function	Description
AVG()	Return the average of non-NULL values.
COUNT()	Return the number of rows in a group, including rows with NULL values.
GROUP_CONCAT()	Return a concatenated string.
MAX()	Return the highest value (maximum) in a set of non-NULL values.
MIN()	Return the lowest value (minimum) in a set of non-NULL values.
STDEV()	Return the population standard deviation.
STDDEV_POP()	Return the population standard deviation.
STDDEV_SAMP()	Return the sample standard deviation.
SUM()	Return the summation of all non-NULL values a set.
VAR_POP()	Return the population standard variance.
VARP_SAM()	Return the sample variance.

For internal reference only not for distribution or sale

VARIANCE()	Return the population standard variance.

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

SELECT Statement

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

The SELECT statement can be used to retrieve (select) data from a table or an expression. It is one of the most complex statement in SQL language with a lot of clauses to perform data retrieval and transformation.

The SELECT statement has the following clauses:

- Select distinct rows using DISTINCT operator.
- Sort rows using ORDER BY clause.
- Filter rows using WHERE clause.
- Select a subset of rows from a table using LIMIT clause.
- Group rows into groups using GROUP BY clause.
- Filter groups using HAVING clause.

Syntax:

```
SELECT [DISTINCT] [columns selection] [FROM <table>] [WHERE
<expression>] [GROUP BY <Expression> [HAVING <Expression>]] [ORDER BY
<sort expression> ASC | DESC] [LIMIT row_to_skip, row_count]
```

Aliasing

Alias is a temporary name given to a table or a column during execution of the query.

Below are the various ways of using an alias.

Column Alias

- Column aliases can be used in the SELECT clause of a SQL query.
- Like all objects, aliases will be in lowercase by default. If mixed-case letters or special symbols, or spaces are required, quotes must be used.
- Column aliases can be used for generated / derived columns.
- Column aliases can be used with GROUP BY and ORDER BY clauses.
- We cannot use a column alias with WHERE and HAVING clauses.

Table Alias

- Table aliases can be used in SELECT lists and in the FROM clause to show the complete record or selective columns from a table.

For internal reference only not for distribution or sale

- Table aliases can be used in WHERE, GROUP BY, HAVING, and ORDER BY clauses.
- When we need data from multiple tables, we need to join those tables by qualifying the columns using table name/table alias.
- The aliases are mandatory for inline queries (queries embedded within another statement) as a data source to qualify the columns in a select list.

Example:

```
SELECT ename employee_name, empno AS employee_id FROM emp;
```

Quoting

Single or Double Quotes

Single quotes are supported by almost all RDBMS as quotation for string literals. MySQL supports both single quote and double quote.

Example:

'hello world'

'2011-07-04 13:36:24' '{1,4,5}'

backticks (` `)

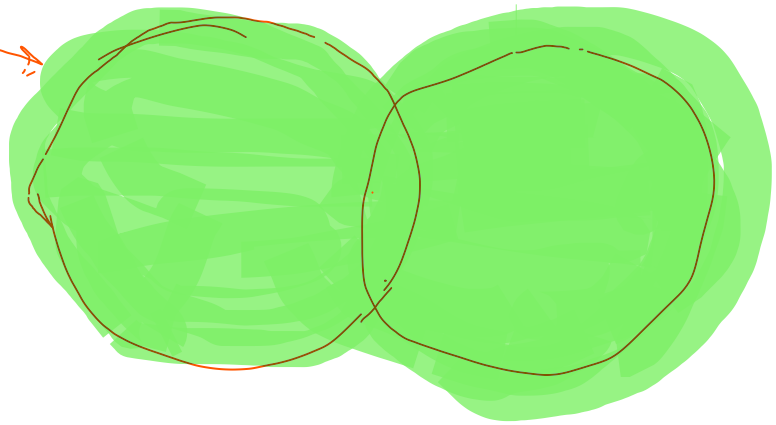
backticks are used to enclose table name or columns names if they are MySQL Keywords or tables names / columns names include whitespaces otherwise no need to use backticks.

```
SELECT * FROM `ORDER`;
```

Table Joins

Table joining technique allows to combine rows from from 2 or more table. Following join types can be used in MySQL

- Cross Join
- Inner Join
- Natural Join
- Left Outer Join
- Right Outer Join
- Full Outer Join
- Self Join



For internal reference only not for distribution or sale

Join Condition

The join condition determines which rows from the two source tables are considered to be a "match". The join condition is specified with the ON or USING clause or implicitly by the word NATURAL.

Cross Join

Cross Join is a cartesian product which contains all rows from first table are joined with all rows from second table. For example if the first table has m rows and second table has n rows then resultant cross joined tables will have m*n rows.

Inner Join

Inner also called as Join combines only matching rows for two tables and leaves out the rows without matches.

Syntax:

```
Select  <columns list>   from tableA A Inner Join TableB  B  On
<condidition>;

Select  <column list>  from TableA A Join TableB B On <condition>;

Select  <column list>  from TableA A Join TableB B Using  <common column>;
```

Natural Join

Natural Join is Inner Join only difference is that there is no need to give any Join Condition.

Syntax:

```
Select  <column list>  from TableA A Natural Join TableB B;
```

Left Outer Join

Left Outer Join also called as Left Join combines all matching rows from both table plus rows from left side table which does not find matches also included.

Syntax:

```
Select  <columns list>   from tableA A Left Outer Join  TableB  B  On
<condidition>;

Select  <column list>  from TableA A Left Join TableB B On <condition>;

Select  <column list>  from TableA A Left Join TableB B Using  <common
column>;
```


For internal reference only not for distribution or sale

Right Outer Join

Right Outer Join also called as Right Join combines all matching rows from both table plus rows from Right side table which does not find matches also included.

Syntax:

```
Select  <columns list>   from tableA A Right Outer Join  TableB  B  On
<condidition>;

Select  <column list>  from TableA A Right Join TableB B On <condition>;

Select  <column list>  from TableA A Right Join TableB B Using  <common
column>;
```

Full Outer Join

Full Outer Join also called as Full Join combines all matching rows from both table plus rows from both left side table and right side table which does not find matches also included.

Syntax

```
Select  <columns list>   from tableA A Full Outer Join  TableB  B  On
<condidition>;

Select  <column list>  from TableA A Full Join TableB B On <condition>;

Select  <column list>  from TableA A Full Join TableB B Using  <common
column>;
```

Note: Currently FULL OUTER JOIN is not supported by MySQL

Self Join

A table can be joined with itself using table & columns aliasing

Combining Queries (Unions)

Results of 2 or more queries can be combined using SET operations like UNION, INTERSECT. Supported set operations in PotgreSQL as follows:

- UNION
- UNION ALL
- INTERSECT
- EXCEPT

For internal reference only not for distribution or sale

UNION

Union set operator allows to combine the results of 2 queries as a single result set.

The following are requirements for the queries in the syntax above:

The number and the order of the columns must be the same in both queries.

The data types of the corresponding columns must be the same or compatible.

Syntax:

```
select  <col1>, <col2>, ... from <table1>  UNION  select <col1>, <col2> ...
from <table2>
```

UNION ALL

UNION ALL works same as UNION but it combines all elements from both the sets where as UNION returns common elements only once

Syntax:

```
select  <col1>, <col2>, ... from <table1>  UNION ALL  select <col1>, <col2>
... from <table2>
```

INTERSECT

Intersect allows to combine results of 2 queries and returns only common rows between them.

Syntax:

```
select  <col1>, <col2>, ... from  <table1>  INTERSECT  select <col1>,
<col2>,..... from <table2>
```

1,2,3,4,5,6

1,2,3,7,8,9

-o/p-> 1,2,3

EXCEPT / MINUS

EXCEPT compares the result sets of two queries and returns the distinct rows from the first query that are not output by the second query.

Syntax: I have used this a lot in Adani - Monthly data load query to check new Plant, Site, Location, Site.

```
select  <col1>, <col2>, ... from  <table1>  EXCEPT  select <col1>,
<col2>,..... from <table2>
```

1,2,3,4,5,6

1,2,3,7,8,9

-o/p-> 4,5,6

Note: Currently MySQL does not support INTERSECT, EXCEPT/MINUS

Sub Queries

A Sub Query is also a query, which is defined under a main query. First Query is also called as outer query or main query and second query is also called as inner query or subquery.

For internal reference only not for distribution or sale

Server first sub query executed based on the sub query value main query will get executed.

If a sub query send single value to its nearest main query then that sub query is called Single-Valued-Sub query.

If a sub query sends multiple values to its nearest main query then that sub query is called Multi-Valued-Sub query.

Note: If a sub query sends multiple values to its nearest main query then we have to use IN or NOT IN operator between Main query and Sub query.

Single valued Subquery:

```
SELECT * FROM TABLENAME WHERE COLUMNNAME =
      (SELECT STATEMENT WHERE CONDITION)
```

Multi valued subquery

A subquery which returns more than one row to the outer query is called as Multi Values subquery. Multivalues values sub queries can be handled by IN,

```
select * from tablename where columname in
(select statement where condition)
```

Example:

display all emp details whose earning salary above avg.salary

```
select * from emp where sal>=(select avg(sal) from emp)
```

```
select * from student where courseid in (select courseid from course where duration > 6)
```

Correlated Sub Query

In Correlated sub queries the outer query and the inner (sub) query are executed simultaneously.

Example:

display max salary for each dept and corresponding emp details

```
select * from emp e where sal=(select max(sal) from emp where deptno=e.deptno)
```


For internal reference only not for distribution or sale

"First outer query gets executed, extracts 1 row at a time(candidate row) and that row is given to inner query for processing, inner query will provide its output to outer query and based on the condition outer query will display the extracted record".

For internal reference only not for distribution or sale

Indexes & Views

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

MySQL Indexes

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

There are 4 kinds of indexes

- Primary Key
- Unique Key
- Index (non-unique)
- Full-Text Index

Primary Key

A primary key is unique and can never be null. It will always identify only one record, and each record must be represented. Each table can only have one primary key.

In InnoDB tables, the table data is arranged in the order of primary key (clustered index) If a primary key does not exists then InnoDB will look for suitable unique key to be used as clustered index and if no unique is present then it will add an invisible column to be used for clustered index.

Adding Primary Key

```
mysql> CREATE TABLE <table name> (....., PRIMARY KEY(column,
[column2]);


mysql> ALTER TABLE <table name>  ADD PRIMARY KEY (column,
[column2]);
```

Note: Primary Key cannot be added with CREATE INDEX statement.

Unique Key

All values in unique index must be unique or NULL. So each key value identifies only one record, but not each record needs to be represented.

For internal reference only not for distribution or sale

Adding Unique Key

```
mysql> CREATE TABLE <table name> (....., UNIQUE KEY(column,
[column2]);

mysql> ALTER TABLE <table name>  ADD UNIQUE (column,
[column2]);

mysql> CREATE UNIQUE INDEX <index name> ON <table name>
(column, [column2]);
```

Non-Unique indexes (Secondary Indexes)

These indexes can be created on any column of table which may contain non-unique values.

Adding Plain Indexes

```
mysql> CREATE TABLE <table name> (....., INDEX (column,
[column2]);

mysql> ALTER TABLE <table name>  ADD INDEX (column,
[column2]);

mysql> CREATE INDEX <index name> ON <table name> (column,
[column2]);
```

Full-Text Indexes

FULLTEXT indexes are used for full-text searches. FULLTEXT indexes are supported only for CHAR, VARCHAR and TEXT columns. Indexing always takes place over the entire column and column prefix indexing is not supported.

Adding Full-Text Indexes

For internal reference only not for distribution or sale

```
mysql> CREATE TABLE <table name> (....., FULLTEXT (column);

mysql> ALTER TABLE <table name>  ADD FULLTEXT (column);

mysql> CREATE FULLTEXT INDEX <index name> ON <table name>
(column);
```

Displaying Indexes

All created indexes information can be queried using SHOW INDEXES FROM or Querying STATISTICS table from information_schema database.

```
mysql> SHOW INDEXES FROM <table name>\G

mysql> SELECT DISTINCT TABLE_NAME, INDEX_NAME FROM
INFORMATION_SCHEMA.STATISTICS WHERE TABLE_SCHEMA =
'<database name';
```

Views

View is a virtual table which exhibits all features of a table but it is actually a saved query which pulls data from one or more tables or other views.

Syntax:

```
CREATE VIEW  <view name>  AS <query >;
```

Example:

```
CREATE VIEW  myview  AS SELECT empno, ename, job, sal, dname, loc FROM emp
join dept ON emp.deptno =  dept.deptno;
```

AUTO_INCREMENT

AUTO_INCREMENT is column attribute which allows to generate a sequence of integer numbers to be saved the columns. This attribute can only be assigned to PRIMARY KEY column if it is having INT datatype.

Syntax:

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

```
CREATE TABLE <tablename> (
   col1  INT AUTO_INCREMENT PRIMARY KEY,
    col2 <datatype>,
   col3 <datatype>
);
```

Example:

```
CREATE TABLE employees (
    emp_no INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50)
);
INSERT INTO employees(first_name,last_name)
VALUES('John','Doe'),
      ('Mary','Jane');
SELECT * FROM employees;
```

emp_no	first_name	last_name
1	John	Doe
2	Mary	Jane

For internal reference only not for distribution or sale

Transactions

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Transactions provide a way of grouping DML statements and before accepting the changes we can review and other accepts them all or discard the changes. Other benefit of the transaction is it protect transaction from concurrently running other transactions and manipulating same data.

MySQL Transaction implementation comes from Storage Engine and currently only InnoDB & NDB Storage engines provides Transaction feature.

How it works

Every transaction starts with START TRANSACTION statement and ends with either COMMIT or ROLLBACK statement.

Syntax:

```
START TRANSACTION;
      some DML Statements;
COMMIT;  / ROLLBACK;
```

Example:

```
START TRANSACTION;
SELECT * FROM EMP;
UPDATE EMP SET SAL = SAL + 100 WHERE ENAME = 'MILLER';
COMMIT;
```

Syntax:

```
START TRANSACTION;
      some DML Statements;
      SAVEPOINT  <savepoint name>;
       some more DML statements;
      ROLLBACK TO SAVEPOINT  <savepoint name>;
COMMIT;  / ROLLBACK;
```


For internal reference only not for distribution or sale

Copying and Moving Data

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Copying Data Between Tables

Data can be copied between tables if they have compatible columns definitions and datatypes.

Copy Data Between Tables

Data can be copied between tables using INSERT statement with a SELECT query in VALUES clause. Data returned by the query will be copied to the given table.

Syntax:

```
INSERT INTO  <table name>  VALUES  (SELECT query);
```

Creating a new table based on select query

We can provide SELECT query while creating the table. New table will have same number of columns as returned by query.

Syntax:

```
CREATE TABLE <table name>  AS  SELECT query;
```

Create new empty table like other table

Syntax:

```
CREATE TABLE <table name> LIKE <other existing table>;
```

Exporting / Importing Data from / to CSV file

Exporting Data

MySQL allows exporting data from a table to a CSV file to TAB separated file in plain text.

Syntax:

```
SELECT  * FROM  <table name> ........

INTO OUTFILE 'C:/tmp/data.csv'

FIELDS ENCLOSED BY '"'

TERMINATED BY ';'

ESCAPED BY '"'

LINES TERMINATED BY '\r\n';
```

Example:

For internal reference only not for distribution or sale

```
SELECT  * FROM  EMP
INTO OUTFILE 'C:/tmp/emp.csv'
FIELDS ENCLOSED BY '"'
TERMINATED BY ';'
ESCAPED BY '"'
LINES TERMINATED BY '\r\n';
```

Importing Data from CSV to table

Data from a CSV file can be imported to a table using LOAD command

Syntax:

LOAD DATA LOCAL INFILE 'c:/tmp/data.csv'

INTO TABLE <table Name>

FIELDS TERMINATED BY ','

ENCLOSED BY '"'

LINES TERMINATED BY '\n'

IGNORE 1 ROWS;

Example:

```
LOAD DATA LOCAL INFILE  'c:/tmp/emp.csv'
INTO TABLE emp2
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n';
```


For internal reference only not for distribution or sale

NoSQL Databases

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

Introduction

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

NoSQL Database

NoSQL is an approach to database design that can accommodate a wide variety of data models, including key-value, document, columnar and graph formats. NoSQL, which stand for "not only SQL," is an alternative to traditional relational databases in which data is placed in tables and data schema is carefully designed before the database is built. NoSQL databases are especially useful for working with large sets of distributed data.

Evolution of NoSQL

The evolution of NoSQL started in 1990's with the development and release of berkeleyDB. Developed by University of California, Berkeley, influenced the development of NoSQL which supported certain applications with specific storage needs.

NoSQL database Classification

NoSQL database is been classified in the following ways.

Key-value stores

Key-value stores, or key-value databases, implement a simple data model that pairs a unique key with an associated value. Because this model is simple, it can lead to the development of key-value databases, which are extremely performant and highly scalable for session management and caching in web applications.

Examples

Aerospike, Berkeley DB, MemchacheDB, Redis and Riak.

Document databases

Document databases, also called document stores, store semi-structured data and descriptions of that data in document format. They allow developers to create and update programs without needing to reference master schema. Use of document databases has increased along with use of JavaScript and the JavaScript Object Notation (JSON)

Examples

Couchbase Server, CouchDB, DocumentDB, MarkLogic and MongoDB.

Wide-column stores

Wide-column stores organize data tables as columns instead of as rows. Wide-column stores can be found both in SQL and NoSQL databases. Wide-column stores can query large data volumes faster than conventional relational databases.

Examples

Google BigTable, Cassandra and HBase.

For internal reference only not for distribution or sale

Graph stores

Graph data stores organize data as nodes, which are like records in a relational database, and edges, which represent connections between nodes. Because the graph system stores the relationship between nodes, it can support richer representations of data relationships.

Examples

AllegroGraph, IBM Graph, Neo4j and Titan

NoSQL classification and Adherence by Vendors

The basic NoSQL database classifications are only guides. Over time, vendors have mixed and matched elements from different NoSQL database family trees to achieve more generally useful systems. That evolution is seen, for example, in MarkLogic, which has added a graph store and other elements to its original document databases. Couchbase Server supports both key-value and document approaches. Cassandra has combined key-value elements with a wide-column store and a graph database. Sometimes NoSQL elements are mixed with SQL elements, creating a variety of databases that are referred to as multimodel databases.

NoSQL vs RDBMS

SQL	NOSQL
RELATIONAL DATABASE MANAGEMENT SYSTEM (RDBMS)	Non-relational or distributed database system.
These databases have fixed or static or predefined schema	They have have dynamic schema
These databases are not suited for hierarchical data storage.	These databases are best suited for hierarchical data storage.
These databases are best suited for complex queries	These databases are not so good for complex queries
Verticlly Scalable	Horizontally scalable

CAP Theorem

The CAP Theorem is a fundamental theorem in distributed database systems that states any distributed system can have at most two of the following three properties.

- Consistency
- Availability

For internal reference only not for distribution or sale

- Partition tolerance

Lets Understand CAP Theorem

The CAP theorem states that a distributed system cannot simultaneously be consistent, available, and partition tolerant.

Distributed databases

Distributed databases are usually non-relational databases that enable a quick access to data over a large number of nodes.

Consistency

Any read operation that begins after a write operation completes must return the latest value.

In a consistent system, once a client writes a value to any server and gets a response, it expects to get that value (or a fresher value) back from any server it reads from.

Availability

Every request receives a (non-error) response – without the guarantee that it contains the most recent write

Partition Tolerance

The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

PACELC theorem

PACELC theorem is an extension to the CAP theorem. It states that in case of network partitioning (P) in a distributed computer system, one has to choose between availability (A) and consistency (C) (as per the CAP theorem), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C)

The purpose of PACELC is to address the oversight in CAP theorem pertaining to consistency/latency tradeoff of replicated systems as it is present at all times during system operation.

Markup Languages

What is JSON?

History

For internal reference only not for distribution or sale

Douglas Crockford is typically credited with discovering the format because he was the first to popularize and specify its use around the year 2001.

Crockford, was developing an AJAX web application framework. However, needed to find a way to enable real-time two-way browser-to-server communication without relying on Flash plugins or Java applets. It was this need that served as the impetus behind the discovery of JSON. As JSON is used to transfer data between browser and server.

JSON consists of nothing more than commas, curly braces, square brackets, and data, it can be easily parsed into an array or object by any programming language.

JSON Data Structures

JSON data is formatted into two data structures that are used universally in all modern programming languages:

- A JSON array is a list of values.
- A JSON object is a collection of name-value pairs.

JSON Arrays:

```
[ "red", "green", "blue", 7 ]
```

JSON Object (Document)

```
{
  name: "Sami"
  Occupation: "Trainer"
  Location: "Hyderabad"
}
```

Complex JSON objects:

```
{
  name: "Sami"
  Occupation: "Trainer"
  Location: "Hyderabad"
  Canteach: ["mongoDB", "MySQL", "PosgreSQL", "Linux"]
  LanguageFluency: {
      English:  "Very Good"
      Hindi: "Very Good"
```


For internal reference only not for distribution or sale

```
      Telugu: "Very Good"
      Tamil: "Can Understand"
  }
}
```

BSON

BSON is a computer data interchange format. The name "BSON" is based on the term JSON and stands for "Binary JSON". It is a binary form for representing simple or complex data structures including associative arrays (also known as name-value pairs), integer indexed arrays, and a suite of fundamental scalar types. BSON originated in 2009 at MongoDB.

MongoDB represents JSON documents in binary-encoded format called BSON behind the scenes. BSON extends the JSON model to provide additional data types, ordered fields, and to be efficient for encoding and decoding within different languages

What is YAML?

Yet Another Markup Language (YAML)

Is a human-readable data-serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored or transmitted. YAML targets many of the same communications applications as XML but has a minimal syntax. It uses both Python-style indentation to indicate nesting, and a more compact format that uses [] for lists and {} for maps making YAML a superset of JSON.

```
employee
    name: "Sami"
    Occupation: "Trainer"
    Location: "Hyderabad"
    Canteach:
        - mongoDB
        - MySQL
        - PosgreSQL
        - Linux
  LanguageFluency:
        English:  "Very Good"
        Hindi: "Very Good"
        Telugu: "Very Good"
```


For internal reference only not for distribution or sale

Tamil: "Can Understand"

For internal reference only not for distribution or sale

Introduction to MongoDB

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

MongoDB is a document oriented database server developed in the C++ programming language. The word Mongo is derived from Humongous.

History

The initial development of MongoDB began in 2007 when the company was building a cloud platform.

MongoDB was developed by a NewYork based organization named 10gen which is now known as MongoDB Inc. It was initially developed as a PAAS (Platform As A Service). Later in 2009, it is introduced in the market as an open source database server that was maintained and supported by MongoDB Inc.

The first ready production of MongoDB has been considered from version 1.4 which was released in March 2010.

Features of MongoDB

- Support ad hoc queries

  In MongoDB, you can search by field, range query and it also supports regular expression searches.

- Indexing

  You can index any field in a document.

- Replication

  MongoDB supports Master Slave replication.

  A master can perform Reads and Writes and a Slave copies data from the master and can only be used for reads or back up (not writes)

- Duplication of data

  MongoDB can run over multiple servers. The data is duplicated to keep the system up and also keep its running condition in case of hardware failure.

- Load balancing

  It has an automatic load balancing configuration because of data placed in shards.

- Supports mapreduce and aggregation tools.

For internal reference only not for distribution or sale

MongoDB Architecture

Client Connection Protocols:

The MongoDB Wire Protocol is a simple socket-based, request-response style protocol. Clients communicate with the database server through a regular TCP/IP socket.

- TCP/IP Socket

  Clients should connect to the database with a regular TCP/IP socket.

- Port

  The default port number for mongod and mongos instances is 27017.

- Default Clients

  The mongo shell is an interactive JavaScript interface to MongoDB. You can use the mongo shell to query and update data as well as perform administrative operations.

  MongoDB Compass is the GUI client for MongoDB.

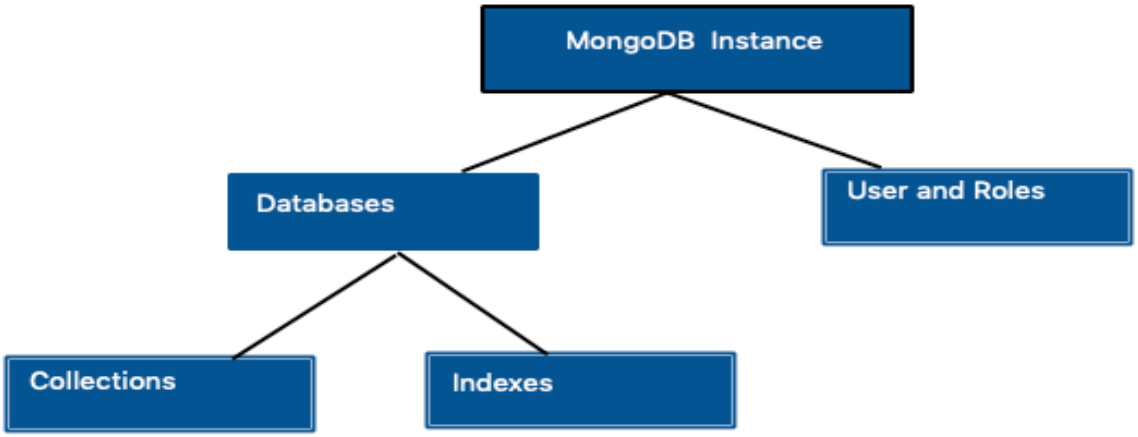
- Security

  MongoDB provides security by implementing user and roles. Security can be disabled.

- MongoDB Query Language

  MongoDB uses its own Query Language which is similar to Javascript and does not uses standard SQL.

Object Hierarchy



For internal reference only not for distribution or sale

A MongoDB instance consisted of 2 high level objects called as DATABASES and USER/ROLES.

DATABASE in turn may have one or more COLLECTIONS and INDEXES which actually stores data.

MONGO DB Vs Traditional RDBMS

RDBMS	MONGODB
ROW	DOCUMENT
TABLE	COLLECTION
COLUMN	PROPERTY or Field
One –Many Mapping	Enbbeded-document's

Data Types

MongoDB supports many datatypes. Some of them are −

String

This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.

Integer

This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.

Boolean

This type is used to store a boolean (true/ false) value.

Double

This type is used to store floating point values.

Min/ Max keys

This type is used to compare a value against the lowest and highest BSON elements.

Arrays

This type is used to store arrays or list or multiple values into one key.

Timestamp

This can be handy for recording when a document has been modified or added.

For internal reference only not for distribution or sale

Object

This datatype is used for embedded documents.

Null

This type is used to store a Null value.

Symbol

This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.

Date

This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.

Object ID

This datatype is used to store the document’s ID.

Binary data

This datatype is used to store binary data.

Code

This datatype is used to store JavaScript code into the document.

Regular expression

This datatype is used to store regular expression.

For internal reference only not for distribution or sale

CRUD Operations

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

CRUD stands for Create, Read, Update and Delete basically they are equivalent to DML operations in RDBMS.

MongoDB Operators

Comparison Operators:

Name	Description
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$in	Matches any of the values specified in an array.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$nin	Matches none of the values specified in an array.
$regex	Matches values as per regular expression given

For internal reference only not for distribution or sale

Logical Operators:

Name	Description
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$not	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
$nor	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.
$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.

Element Operators

Name	Description
$exists	Matches documents that have the specified field.
$type	Selects documents if a field is of the specified type.

For internal reference only not for distribution or sale

The Crud Operations

insert Method

Inserts a document or documents into a collection.

Syntax:

```
db.collection.insert([
   {doc1},{doc2},....],
   {
     writeConcern: <document>,
     ordered: <boolean>
   }
)
```

Examples:

```
db.products.insert( { item: "card", qty: 15 } )
```

```
db.products.insert(
   [
     { _id: 11, item: "pencil", qty: 50, type: "no.2" },
     { item: "pen", qty: 20 },
     { item: "eraser", qty: 25 }
   ]
)
```

Unordered Inserts:

```
db.products.insert(
   [
     { _id: 20, item: "lamp", qty: 50, type: "desk" },
     { _id: 21, item: "lamp", qty: 20, type: "floor" },
     { _id: 22, item: "bulk", qty: 100 }
   ],
   { ordered: false }
)
```


For internal reference only not for distribution or sale

Inserts with Write Concerns:

```
db.products.insert(
     { item: "envelopes", qty : 100, type: "Clasp" },
     { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

insertOne method

Inserts a single document into collection

```
db.collection.insertOne(
   <document>,
   {
      writeConcern: <document>
   }
)
```

insertMany Method

Inserts multiple documents into the collection

```
db.collection.insertMany(
   [ <document 1> , <document 2>, ... ],
   {
      writeConcern: <document>,
      ordered: <boolean>
   }
)
```

find Method

For internal reference only not for distribution or sale

Selects documents in a collection or view and returns the selected documents.

Syntax:

```
db.collection.find({query}, {projection})
```

query: is a condition to match the documents

Projection: Selection of fields to return. Projection cannot contain both exclude and include specification except for _id field

Query Examples:

```
db.bios.find( { qty: 23 } )
db.bios.find( { "name.last": "Hopper" } )
```

Note: To access fields in an embedded document, use dot notation **("<embedded document>.<field>")**.

Using `$in` Operator

```
db.employees.find(
   { _id: { $in: [ 1001, 1005) ] } }
)
```

Using $in Operator on Array to match any of the element

```
db.employees.find(
   { canteach: { $in: [ "mongodb", "MySQL") ] } }
)
```

Using Regex with find method

For internal reference only not for distribution or sale

```
db.employees.find(
   { name: { $regex: /^N/ } }
)
```

Using comparison Operators

```
db.employees.find( { doj: { $gt: new Date('2015-01-01') } } )
db.employees.find( { salary: { $gt: 10000 } } )
db.employees.find( { salary: { $lt: 10000 } } )
```

Range Queries:

```
db.employees.find( { salary: { $gte: 10000, $lt: 20000 } } )
db.employees.find( { doj: { $gt: new Date('2015-01-01'), $lt: new
Date('2017-12-31') } } )
```

Queries on Multiple fields

```
db.employees.find( {
   doj: { $gt: new Date('1920-01-01') },
   salary: { $gt : 10000}
} )
```

Querying for documents without a particular field

```
db.employees.find( {
   doj: { $exists: false }
} )
db.employees.find( {
   doj: null
} )
```

Querying an Array

```
db.employees.find( { canteach: { $all: [ "mongodb", "mysql" ] } } )

db.employees.find(
   { canteach: { $in: [ "mongodb", "MySQL") ] } }
)

db.employees.find( { canteach: { $size: 2} } )
```


For internal reference only not for distribution or sale

$elemmatch : The **$elemMatch** operator matches documents that contain an array field with at least one element that matches all the specified query criteria

Consider the following collection

```
{ _id: 1, results: [ 82, 85, 88 ] }
{ _id: 2, results: [ 75, 88, 89 ] }
```

```
db.scores.find(
   { results: { $elemMatch: { $gte: 80, $lt: 85 } } }
)
```

Specify the Fields to Return (projection)

```
db.employees.find( {}, {_id:0,  name:1, salary:1 } } )
```

Sorting Result

Sorting the Result (sort() method)

```
db.employees.find().sort({name: 1})
```

Limiting output

Limit method limit()

```
db.employees.find().limit(5)
```

Skip method

```
db.employees.find().skip(5)
```

Updating Data

Update Method

```
db.collection.update(query, update, options)
```

Modifies an existing document or documents in a collection. The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the update parameter.

Syntax:

For internal reference only not for distribution or sale

```
db.collection.update(
   <query>,
   <update>,
   {
     upsert: <boolean>,
     multi: <boolean>,
     writeConcern: <document>,

   }
)
```

Upsert Behavior

If upsert is true and no document matches the query criteria, update() inserts a single document. The update creates the new document.

If upsert is true and there are documents that match the query criteria, update() performs an update.

```
db.people.update(
   { name: "Andy" },
   {
      name: "Andy",
      rating: 1,
      score: 1
   },
   { upsert: true }
)
```

Updating Multiple Documents

If multi is set to true, the update() method updates all documents that meet the <query> criteria.

```
db.books.update(
   { stock: { $lte: 10 } },
```


For internal reference only not for distribution or sale

```
   { $set: { reorder: true } },
   { multi: true }
)
```

With write concern

```
db.books.update(
   { stock: { $lte: 10 } },
   { $set: { reorder: true } },
   {
     multi: true,
     writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

Changing Data

Changes to documents can be done using any of the **$set** or **$inc** Operators

Example of **$set**:

```
db.books.update(
   { _id: 1 },
   {
     $set: {
       item: "ABC123",
       "info.publisher": "2222",
       tags: [ "software" ],
       "ratings.1": { by: "xyz", rating: 3 }
     }
   }
)
```

Example of **$inc** operator:

```
db.books.update(
   { _id: 1 },
   { $inc: {stock:5}
```


For internal reference only not for distribution or sale

```
)
```

Remove Fields

Remove Fields from a Document

The following operation uses the $unset operator to remove the field

```
db.employees.update( { _id: 1 }, { $unset: { salary: 1 } } )
```

updateOne method

Updates one document which matches the filter

```
db.collection.updateOne(
   <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>,
   }
)
```

updateMany method

Updates multiple documents within the collection based on the filter.

Syntax:

```
db.collection.updateMany(
   <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>,
   }
)
```


For internal reference only not for distribution or sale

Deleting Data

Remove Method

Remove method removes / deletes the documents matching given query.

Syntax:

```
db.collection.remove(
   {<query>},
   {
     justOne: <boolean>,
     writeConcern: <document>,
   }
)
```

And empty query deletes all documents from the collections. Example below.

```
db.employees.remove( { } )
```

remove method with a query deletes all the matching document

```
db.employees.remove({name:"Sami"})
```

Remove method with justOne deletes first match

```
db.employees.remove({name:"Sami"}, {justOne: true})
```

Delete Method

Like remove method delete method also deletes the documents matching given query. There are 2 variations namely deletemany and deleteone.

```
db.collection.deleteMany(
   <filter>,
   {
      writeConcern: <document>,
      collation: <document>
   }
)
```

```
db.collection.deleteOne(
```


For internal reference only not for distribution or sale

```
   <filter>,
   {
      writeConcern: <document>,
      collation: <document>,
      hint: <Index hint>
  }
)
```

Advance Collection Creation

MongoDB is schema less, means there is no fixed schema or field definitions for collections however MongoDB allow to specify certain validations such as size limit for collection or required fields.

Creating capped collection

Creating capped collection requires us to provide SIZE and MAX values.

SIZE: Maximum size of the collections. When Collection reaches this size then the oldest documents are deleted to make room for new documents. MongoDB ensures that the collection does not grow above the mention size. Mentioning SIZE is must for capped collections.

MAX: specifies maximum number of document a collection can have. SIZE takes the precedence when SIZE limit is reached before reaching MAX.

Example:

```
db.createCollection("stock", { capped : true, size : 5242880, max : 5000 } )
```

The above example creates a collection named stock with 5MB size cap.

Collection Validations

MongoDB does not enforce fixed schema requirements but it provides validations to declare required field list and field properties like datatype, input validations etc..

Required Fields

Creating a collections with required fields

```
db.createCollection( "employees", {
   validator: { $jsonSchema: {
      bsonType: "object",
      required: [ "ename", "job", "salary" ] }
```


For internal reference only not for distribution or sale

```
    }
  }
)
```

A new collection named "employees" is created where every document required to have "ename", "job" and "salary" columns.

Input Validation

Schema validations allows to specify datatype and other input validations for documents in a collection. properties object which is embedded object within validator object facilitates input validations.

```
    properties: {
        field1 : {
           bsonType: "<type>",
            enum : [v1,v2,....],
           description: "<some description>"
        },
        field2: {
           bsonType: "<type>",
            minimum : <number>,
            maximum : <number>,
           description: "<some descriptions>"
        }
     }
```

Example:

```
db.createCollection( "employee2", {
   validator: { $jsonSchema: {
      bsonType: "object",
      required: [ "ename", "job", "salary" ],
      properties: {
         ename : {
            bsonType: "string",
            description: "Name of the Employee"
         },
         job: {
            bsonType: "string",
```


For internal reference only not for distribution or sale

```
                enum : ["Manager", "Analyst","Clerk","Salesman"],
              description: "Designation of Employee, Choose one of the
mentioned jobs"
          },
          salary: {
             bsonType: "decimal",
              minimum : 1000,
                maximum : 1000000,
           description: "Salary of Employee"
          },
          deptno: {
          bsonType: "int",
          description: "Department number of Employee"
          }
       }
       }
     }
   }
)
```

View Collection Validation Specification use following method.

```
db.getCollectionInfos()
```


For internal reference only not for distribution or sale

Aggregation

by :- Mohammed Abdul Sami
For internal reference only not for distribution or sale

The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into aggregated results.

Syntax:

```
db.collection.aggregate([ {stage1}, {stage2}, {stage3},....])
```

An aggregation pipeline consists of one or more stages that process documents:

- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

Aggregation stages

$match stage

Usage:

```
{ $match: { <query> } }
```

Example:

```
db.emp.aggregate([{$match: {job:"CLERK"}}])
```

In Aggregation **$match** stage can come at any position.

$group Stage

Group stage perform group level aggregation computations like sum, count, min, max etc. For group level computations it further needs accumulators as shown in table below.

Name	Description
$avg	Returns an average of numerical values. Ignores non-numeric values.
$first	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.

For internal reference only not for distribution or sale

$last	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
$max	Returns the highest expression value for each group.
$min	Returns the lowest expression value for each group.
$stdDevPop	Returns the population standard deviation of the input values.
$stdDevSamp	Returns the sample standard deviation of the input values.
$sum	Returns a sum of numerical values. Ignores non-numeric values.

Grouping Examples:

Sum of Salary for each job title:

```
db.emp.aggregate([ {$group:{ _id:"$job", tot_sal:{$sum:"$sal"}}}])
```

Number of employees per job title:

```
db.emp.aggregate([ {$group:{ _id:"$job", No_OF_Emp:{$sum:1}}}])
```

Total No. of employees:

```
db.emp.aggregate([ {$group:{ _id:null, All_Emp_COUNT:{$sum:1}}}])
```

Lowest Salary:

```
db.emp.aggregate([ {$group:{ _id:null, All_Emp_COUNT:{$min: "$sal" }
}}])
```

Highest Salary:

```
db.emp.aggregate([ {$group:{ _id:null,
All_Emp_COUNT:{$max:"$sal"}}}])
```


For internal reference only not for distribution or sale

$bucket stage

Syntax:

```
{
  $bucket: {
      groupBy: <expression>,
      boundaries: [ <lowerbound1>, <lowerbound2>, ... ],
      default: <literal>,
      output: {
         <output1>: { <$accumulator expression> },
         ...
         <outputN>: { <$accumulator expression> }
      }
   }
}
```

Lets say that we want categorized products according to salePrice and count number of products within each price range.

Here the price range we are using is called as boundaries based on which grouping is done for salePrice field

Example:

```
db.emp.aggregate([
{$bucket:{groupBy:"$sal",boundaries:[0,1000,2000,5000],default:"Other
",output:{"count":{$sum:1}}}}])
```

Example of bucket with 2 output fields fist will count no. of products in each bucket and second will report what is the maximum salePrice in each bucket

```
db.emp.aggregate([
{$bucket:{groupBy:"$sal",boundaries:[0,1000,2000,3000,4000,5000],defa
ult:"Other",output:{"count":{$sum:1},"max":{$max:"$sal"}}}}])
```

$bucketAuto stage

Here it will automatically decides on the boundaries and produce result

syntax:

For internal reference only not for distribution or sale

```
{
  $bucketAuto: {
      groupBy: <expression>,
      buckets: <number>,
      output: {
         <output1>: { <$accumulator expression> },
         ...
      }
  }
}
```

Example:

```
db.emp.aggregate([
{$bucketAuto:{groupBy:"$sal",buckets:4,output:{"count":{$sum:1},"max"
:{$max:"$sal"}}}}])
```

$addFields stage

Adds new fields to documents. $addFields outputs documents that contain all existing fields from the input documents and newly added fields

Syntax:

```
{ $addFields: { <newField>: <expression>, ... } }
```

Lets add a new field called Annual Salary which is computed 12 times of sal.

```
db.emp.aggregate([ {$addFields:
{Annual_Salary:{$multiply:["$sal",12]}}}])
```

$project stage

Passes along the documents with the requested fields to the next stage in the pipeline. The specified fields can be existing fields from the input documents or newly computed fields.

Syntax:

```
{ $project: { <specification(s)> } }
```

Example:

```
db.emp.aggregate([ {$project: {_id:0,ename:1,job:1, sal:1}}])
```

$project and $addField Together

For internal reference only not for distribution or sale

```
db.emp.aggregate([ {$addFields: {Ann_Sal:{$multiply:["$sal",12]}}},
{$project: {_id:0,ename:1,job:1, sal:1, Ann_Sal:1}} ])
```

$lookup stage

Performs a left outer join to an un-sharded collection in the same database to filter in documents from the “joined” collection for processing.

Syntax:

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

For example if we have a dept collection with deptno, name and location filelds and an employees collection with a matching deptno filed we can match both the collections.

```
db.dept.aggregate([{ $lookup: {
from:  "emp",
localField: "_id",
foreignField: "deptno",
as:  "employee-details"
}
}
]
)
```


For internal reference only not for distribution or sale