# SQL Concepts

A **database** is a container that holds tables and other SQL structures related to those tables.

A **column** is a piece of data stored by our table.

A **row** is a single set of columns that describe attributes of a single thing.

Columns and rows together make up a table

The word **field** is often used instead of column

The word **record** is often used instead of row

## CREATE DATABASE

**CREATE DATABASE table\_name**;

**USE DATABASE table\_name**;

Capitalization and underscores help our program in SQL (even though SQL doesn’t need them)

## CREATE TABLE

**CREATE TABLE table\_name (column\_name DATATYPE, … )**

## DATATYPES

CHAR(CHARACTER)

It’s a rigid and prefers its data to be a set length

INT(INTEGER)

The number should be whole and it can be negative too

BLOB

It takes large amount of text data

DATE

It keeps track of our dates and time

VARCHAR

It holds text data of up to 255 characters in length

DATETIME or TIMESTAMP

Keeps track of the dates

DEC(DECIMAL)

It gives numbers with decimal places

### DEC(3,2)

The first argument is the total number of digits of precision, and the second argument is the number of digits after the decimal point.

## DESCRIBE

**DESC table\_name**

Describes the structure of the table

## DROP TABLE

**DROP TABLE table\_name**

It deletes the table and any data in it

## INSERT DATA

**INSERT INTO table\_name(column name, …)**

**VALUES(‘value1’, …)**

**INSERT INTO table\_name**

**VALUES(‘value1’, …)**

### INSERTING DATA

Insert data with single quotes on it.

**Handle quotes with a backlash**

INSERT INTO table\_name VALUES(‘Grover**\’s** MILL’, ‘NJ’)

**Handle quotes with an extra single quote**

INSERT INTO table\_name VALUES(‘Grover**’’s** MILL’, ‘NJ’)

## INNER NULL

CREATE TABLE table\_name (column\_name DATATYPE **NOT NULL**, …)

## DEFAULT

CREATE TABLE table\_name (column\_name **DEFAULT 1.00**)

Using a **default** value fills the empty columns with a specified value

## SELECT

**SELECT \* from table\_name**

Select all the data from all the columns from table\_name

### WHERE

SELECT \* FROM table\_name **WHERE column\_name = ‘column\_data’**

Select all the data from the selected column where that column consists of selected data

The **VARCHAR, CHAR, BLOB, DATA** and **TIME** data types need single quotes. The numeric types **DEC** and **INT**, do not

### SPECIFIC COLUMN

**SELECT column\_name, column\_name, … FROM table\_name**

### BOOLEAN

#### AND

SELECT column\_name FROM table\_name WHERE column\_name = ‘column\_data’

**AND** column\_name = ‘column\_data’

#### OR

SELECT column\_name FROM table\_name WHERE column\_name = ‘column\_data’

**OR** column\_name = “column\_data’

#### NOT

SELECT column\_name FROM table\_name **WHERE column\_name NOT IN (‘column\_data\_1’, ‘column\_data\_2’)**

SELECT column\_name FROM table\_name **WHERE NOT column\_name BETWEEN number\_1 and number\_2**

SELECT column\_name FROM table\_name **WHERE NOT column\_name LIKE ‘A%’**

### COMPARISON OPERATOR

#### Greater than >

SELECT column\_name FROM table\_name column\_name = ‘column\_data’ AND

column\_name **>** column\_value

#### Less than <

SELECT column\_name FROM table\_name column\_name = ‘column\_data’ AND

column\_name **<** column\_value

#### Less than and Equals <=

SELECT column\_name FROM table\_name column\_name = ‘column\_data’ AND

column\_name **<=** column\_value

#### Greater than and Equals >=

SELECT column\_name FROM table\_name column\_name = ‘column\_data’ AND

column\_name **>=** column\_value

### NULL

SELECT column\_name FROM table\_name WHERE column\_name **IS NULL**

### LIKE

It is a wildcard operator

SELECT \* FROM table\_name WHERE column\_name **LIKE ‘%CA’**

The percent is a stand-in for any number of unknown characters

SELECT \* FROM table\_name WHERE column\_name **LIKE ‘\_CA’**

The underscore is a stand-in for just one unknown character

### BETWEEN

SELECT column\_table FROM table\_name WHERE column\_name **BETWEEN number\_value\_1 and number\_value\_2**

### IN

SELECT column\_table FROM table\_name WHERE **column\_name IN (‘data\_1’, ‘data\_2’)**

## DELETE

We can use WHERE clauses with DELETE statements the same way we use them with INSERT statements

**DELETE FROM table\_name WHERE table\_column = ‘table\_data’**

## UPDATE

**UPDATE table\_name SET column\_name = ‘column\_data’ WHERE column\_name = ‘column\_data’**

Update statements can replace DELETE/INSERT combinations. Update statements can be used on multiple records in our table. We can use them with basic math operators to manipulate our numeric values

UPDATE table\_name **SET column\_name = column value + 1**

WHERE column\_name = ‘column\_data’

OR column\_name = ‘column\_data’

UPDATE table\_name **SET column\_name = RIGHT(another\_column\_name, 2)**

## ATOMIC DATA

A little piece of information that can’t or shouldn’t be divided. Data should be broken into the smallest pieces of data that can’t or shouldn’t be divided.

**Rule 1:** A column with atomic data can’t have several values of the same type of data in that column

**Rule 2:** A table with atomic data can’t have multiple columns with the same type of data.

## NORMAL

Making our atomic is the first step in creating a NORMAL table.

**Benefit 1:** Normal tables won’t have duplicate data, which will reduce the size of our database

**Benefit 2:** With less data to search through our queries will be faster

Each row of data must contain atomic values

Each row of data must have a unique identifies, known as a PRIMARY KEY

## PRIMARY KEY

A primary key is a column in our table that makes each record unique.

**Rule 1:** A primary key can’t be NULL

**Rule 2:** A primary key must be given a value when the record is inserted

**Rule 3:** The primary must be compact

**Rule 4:** The primary key values can’t be changed

CREATE TABLE table\_name(

**column\_name\_1 DATATYPE NOT NULL AUTO\_INCREMENT,**

column\_name\_2 DATATYPE,

**PRIMARY KEY (column\_name\_1)**

)

Auto Increments just automatically fill the column with value that starts on row 1 with a value of 1 and goes up in increments of 1.

## SHOW

**SHOW CREATE TABLE table\_name**

## BACKTICK

It allows us to use whatever field name we wish when designing your table. Sometimes it makes a lot of sense to name a field `key`, `order`, or `values`... all of which require backticks when referring to them.

## ALTER

CHANGE: Changes both the name and data type of an existing column

MODIFY: Changes the data type or position of an existing column

ADD: Add a column to our table and it needs data type

DROP: Drop the column from the table.

### RENAME

ALTER TABLE **old\_table\_name RENAME TO new\_table\_name**

### CHANGE

ALTER TABLE table\_name **CHANGE COLUMN old\_column\_name new\_column\_name** INT NOT NULL AUTO\_INCREMENT, **ADD PRIMARY KEY** **(new\_column\_name)**

ALTER TABLE table\_name **CHANGE** **COLUMN old\_column\_name new\_column\_name** **NEW\_DATA\_TYPE**

### MODIFY

ALTER TABLE table\_name **MODIFY COLUMN column\_name DATATYPE**

### ADD

ALTER TABLE table\_name **ADD COLUMN column\_name DATATYPE AFTER specific\_column\_name**

**ALTER TABLE table\_name ADD COLUMN column\_name DATATYPE FIRST**

### DROP

ALTER TABLE table\_name **DROP COLUMN column\_name**

## STRING FUNCTION

Text values and values stored in CHAR or VARCHAR columns are known as strings. String functions allow us to select part of a text column.

We can use string functions in combination with SELECT, UPDATE, and DELETE

### RIGHT

SELECT **RIGHT(column\_name, count\_number\_from\_right)** FROM table\_name

### SUBSTRING

SELECT **SUBSTRING(COLUMN\_NAME, start\_position, length)** FROMtable\_name

### SUBSTR

UPDATE table\_name SET column\_name = **SUBSTR(column\_name\_1, LENGTH(column\_name\_2) + 2)**

### SUBSTRING INDEX

SELECT **SUBSTRING\_INDEX(column\_name, ‘delimiter’, which\_one)** FROM table\_name

**which\_one,** if it is “1”, it means it is looking for the first comma. If it were “2” it would keep going until finds a second comma and grab everything in front of that.

### UPPER

SELECT **UPPER(column\_name)** FROM table\_name

### LOWER

SELECT **LOWER(column\_name)** FROM table\_name

### LTRIM

SELECT **LTRIM(column\_name)** FROM table\_name

Removes any string with extra spaces removed from the left

### RTRIM

SELECT **RTRIM(column\_name)** FROM table\_name

Removes any string with extra spaces removed from the right

### REVERSE

SELECT **REVERSE(column\_name)** FROM table\_name

Reverses the order of letters in the string

### LENGTH

SELECT **LENGTH(column\_name)** FROM table\_name

Returns a count of hominy characters are in the string

## CONDITIONS

UPDATE table\_name

SET column\_name =

**CASE**

**WHEN column\_name\_1 = some\_value\_1**

**THEN new\_value\_1**

**WHEN column\_name\_2 = some\_value\_2**

**THEN new\_value\_2**

**ELSE new\_value\_3**

**END**

## ORDER BY

SELECT column\_name FROM table\_name WHERE column\_name = ‘column\_data’ **ORDER BY column\_name**

It allows us to alphabetically order any columns

### ORDER TWO COLUMNS

**SELECT** **column\_name\_1, column\_name\_2 FROM table\_name** **ORDER BY order\_column\_name\_1, order\_column\_name\_2**

column\_name\_1 will get ordered by order\_column\_name\_1

column\_name\_2 will get ordered by order\_column\_name\_2

### ORDER WITH MULTIPLE COLUMNS

**SELECT \* FROM table\_name ORDER BY order\_column\_name\_1, order\_column\_name\_2, order\_column\_name\_3**

First the columns are ordered by order\_column\_1 then order\_column\_2 and at the end order\_column\_3

We can sort by as many columns as we need

### SQL RULES OF ORDER

Non-alphabet characters show up before and after numbers.

Numbers show up before text characters.

NULL values show up before numbers and alphabet characters

Uppercase characters show up before and lowercase letters

A 1 will show up before A1

Order: **!= & ( \* + - ? @ ~**

### DESC

Keyword DESC after the column name in ORDER BY clause reverses the order of the result

SELECT \* FROM table\_name ORDER BY order\_column\_name\_1 **DESC**

### ASC

SELECT \* FROM table\_name ORDER BY order\_column\_name\_1 **ASC**

We can put ASC but it is not the necessary since the default order is ASC

## ARITHMETIC FUNCTION

### SUM

SELECT **SUM(column\_name)** FROM table\_name WHERE another\_column\_name = “column\_data”

### GROUP BY

SELECT **column\_name\_1,** **SUM(column\_name\_2)** FROM table\_name **GROUP BY column\_name\_1 ORDER BY SUM(column\_name\_2) DESC**

### AVERAGE

SELECT **column\_name\_1,** **AVG(column\_name\_2)** FROM table\_name **GROUP BY column\_name\_1**

### MIN AND MAX

SELECT **column\_name\_1,** **MAX(column\_name\_2)** FROM table\_name **GROUP BY column\_name\_1**

SELECT **column\_name\_1,** **MIN(column\_name\_2)** FROM table\_name **GROUP BY column\_name\_1**

### COUNT

SELECT **COUNT(column\_name)** FROM table\_name

It will return the number of rows in a column

### DISTINCT

SELECT **DISTINCT(column\_name)** FROM table\_name ORDER BY **another\_column\_name**

SELECT **COUNT(DISTINCT column\_name)** FROM table\_name

### LIMIT

SELECT **column\_name FROM table\_name LIMIT 2**

Lets only the first two data to show up

SELECT **column\_name FROM table\_name LIMIT 0,5**

Let us see the data from row 0 to row 5

## COMMENT

**/\*Select all the columns**

**of all the records**

**in the Customers table:\*/**

**--SELECT \* FROM Customers;**

## SCHEMA

A descript of the data (the columns and tables) in our database, along with any other related objects and the way they all connect is known as **schema**

Creating a diagram of our table lets us keep the design of the table separate from the data that inside of it.

## FOREIGN KEY

The **foreign key** is a column in a table that references the **primary key** of another table

We will be able to insert values into our foreign key that exist in the table the key came from, the parent table. This is called **referential integrity.**

Creating a foreign key as a constraint in our table give us definite advantages. We’ll get errors if we violate the rules, which will stop us accidentally doing anything to break the table.

Example:

**CREATE TABLE interest (**

--Adding the primary key command to the line where we set it up is quicker way to designate

**int\_id INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,**

**interest VARCHAR(50) NOT NULL,**

**contact\_id INT NOT NULL,**

--The key came from (my\_contacts) 🡪 contact\_id and that it’s a foreign key

**CONSTRAINT my\_contacts\_contact\_id\_fk**

**FOREIGN KEY (contact\_id)**

**REFERENCES my\_contact (contact\_id)**

## RELATIONSHIP BETWEEN TABLES

|  |  |
| --- | --- |
|  | One to One |
|  | One to Many |
|  | Many to Many |

### ONE-TO-ONE

**TABLE B**

**TABLE A**

**ONLY ONE** of these rows -- Matches TO -- **ONLY ONE** of these rows

One-to-One: Exactly one row of a parent table is related to one row of child table

### ONE-TO-MANY

**TABLE B**

**TABLE A**

**ONLY ONE** of these rows -- Matches TO -- **MANY** of these rows

One-to-Many: A record in TABLE A can have many matching records in TABLE B, but a record in TABLE B can only match one record in TABLE A

### MANY-TO-MANY

**TABLE B**

**TABLE A**

**MANY** of these rows -- Matches TO -- **MANY** of these rows

### JUNCTION TABLE

Many-to-Many 🡪 A junction table holds a key from each table

## NORMAL FORM REVISITED

### ATOMIC DATA

A little piece of information that can’t or shouldn’t be divided. Data should be broken into the smallest pieces of data that can’t or shouldn’t be divided.

**Rule 1:** A column with atomic data can’t have several values of the same type of data in that column

**Rule 2:** A table with atomic data can’t have multiple columns with the same type of data.

### NORMAL

Making our atomic is the first step in creating a NORMAL table.

**Benefit 1:** Normal tables won’t have duplicate data, which will reduce the size of our database

**Benefit 2:** With less data to search through our queries will be faster

Each row of data must contain atomic values

Each row of data must have a unique identifies, known as a PRIMARY KEY

### FIRST NORMAL FORM

**1NF**

**Rule 1:** Columns contain only atomic values

**Rule 2:** No repeating groups of data

A key made of two or more columns is known as a **composite key**. A composite key is a primary key composed of multiple columns, creating a unique key. When a column’s data must change when another column’s data is modified

When a column’s data must change when another column’s data is modified, the first column is **functionally dependent** on the second.

**{Name + Power} 🡺 Composite Key**

|  |  |  |
| --- | --- | --- |
| **Name** | **Power** | **Weakness** |
| Superman | Fly | Dumb |
| Batman | Rope | Gadgets Dependent |
| Spiderman | Web | Monster |

**Shorthand Notations**

**T.x ->; T.y**

Technical term for this a shorthand notation

Relational Table is called T

Column y is functionally dependent on column x

**Partially functional dependency:** It means that a non-key column is dependent on some, but not all, of the columns in a composite primary key.

**{Name + Power} 🡺 Composite Key {Name is dependent 🡪 Initials}**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Power** | **Weakness** | **Initials** |
| Superman | Fly | Dumb | SM |
| Batman | Rope | Gadgets Dependent | BM |
| Spiderman | Web | Monster | SM |

**Transitive function dependency:** If changing any of the non-key columns might cause any of the other columns to change, we have a transitive dependency.

When any non-key column is related to any of the other non-key columns.

arch\_enemey\_id 🡺 arch\_enemy\_city

|  |  |  |
| --- | --- | --- |
| **Name** | **arch\_enemy\_id** | **arch\_enemy\_city** |
| Superman | 4 | La La City |
| Batman | 5 | Arkham City |
| Spiderman | 8 | Times City |

### SECOND NORMAL FORM

**2NF**

**Rule 1:** Be in 1NF

**Rule 2:** Have no partial functional dependencies

Your 1NF table is also 2NF, if all the columns in the table are part of the primary key or It has a single column primary key.

### THIRD NORMAL FORM

**3NF**

**Rule 1:** Be in 2NF

**Rule 2:** Have no transitive dependencies

If your table has an artificial primary key and no composite primary key, it’s in 2NF

### SUMMARY NORMAL FORM

1. A column with atomic data can’t have several values of the same type of data in that column
2. A table with atomic data can’t have multiple columns with the same type of data.
3. No repeating groups of data
4. Have no partial functional dependencies
5. Have no transitive dependencies

## MULTIPLE QUERIES

### CREATE TABLE, THEN INSERT WITH SELECT

CREATE TABLE profession (id INT(11) NOT NULL AUTO\_INCREMENT PRIMARY KEY, profession varchar(20))

* Fill up the **profession column** of the **profession table** with the values from our **SELECT**

**INSERT INTO profession (profession)**

new\_**table new\_column**

**SELECT profession FROM my\_contacts**

existing\_column existing\_table

**GROUP BY profession**

**ORDER BY profession**

### CREATE TABLE WITH SELECT, THEN ALTER TO ADD PRIMARY KEY

* Create the **profession table** with **one column**, full of the values from the **SELECT**

**CREATE TABLE profession AS**

**SELECT profession FROM my\_contacts**

**GROUP BY profession**

**ORDER BY profession**

ALTER TABLE profession

ADD COLUMN id INT NOT NULL AUTO\_INCREMENT FIRST,

ADD PRIMARY KEY(id)

### CREATE, SELECT AND SELECT

* Create the profession table with both a primary key and a profession column, and fill the profession column with the values from the **SELECT**

**CREATE TABLE profession(**

**id INT(11) NOT NULL AUTO\_INCREMENT PRIMARY KEY,**

**profession varchar(20)**

**) AS**

**SELECT profession FROM my\_contacts**

**GROUP BY profession,**

**ORDER BY profession**

## ALIASES

Table aliases are also called **correlation names**

SELECT profession **AS mcprof**

FROM my\_contacts **AS mc**

**GROUP BY mc\_prof**

**ORDER BY mc\_prof**

Removing AS makes no difference but it means the same thing

SELECT profession **mcprof**

FROM my\_contacts **mc**

**GROUP BY msc\_prof**

**ORDER BY mc\_prof**

## CARTESIAN JOIN

The **cross-join** returns from one table crossed with every row from the second

**boy**

Davey

Bobby

**toy**

hula hoop

balsa glider

toy soldiers

harmonica

baseball cards

2\*5 = 10 combinations

|  |  |
| --- | --- |
| **toy** | **boy** |
| hula hoop | Davey |
| balsa glider | Bobby |
| toy soldiers | Davey |
| harmonica | Bobby |
| baseball cards | Davey |
| hula hoop | Bobby |
| balsa glider | Davey |
| toy soldiers | Bobby |
| harmonica | Davey |
| baseball cards | Bobby |

**SELECT t.toy, b.boy**

**FROM toy AS t**

**CROSS JOIN**

**boys as b**

### INNER JOIN

An **inner join** is a cross join with some result rows removed by a condition in the query.

An inner join combines the records from two tables using comparison operators in a condition

**SELECT column\_name FROM table\_1**

**INNER JOIN table\_2 ON some\_condition**

**Table Boys**

|  |  |  |
| --- | --- | --- |
| **boy\_id** | **boy** | **toy\_id** |
| 1 | Davey | 2 |
| 2 | Bobby | 1 |
| 3 | Beaver | 2 |

**Table Toys**

|  |  |
| --- | --- |
| **toy\_id** | **toy** |
| 1 | hula hoop |
| 2 | balsa glider |
| 3 | toy soldiers |

EQUIJOIN INNER JOIN

Test for equality

**SELECT boys.boy, toys.toy FROM boys**

**INNER JOIN toys**

**ON boys.toy\_id = toys.toy\_id**

**Result**

|  |  |
| --- | --- |
| **boy** | **Toy** |
| Beaver | balsa Glider |
| Bobby | hula hoop |
| Davey | balsa glider |

### NON EQUIJOIN INNER JOIN

test for inquality

**SELECT boys.boy, toys.toy FROM boys**

**INNER JOIN toys**

**ON boys.toy\_id <> toys.toy\_id**

**Result**

|  |  |
| --- | --- |
| **boy** | **Toy** |
| Beaver | hula hoop |
| Beaver | toy soldiers |
| Bobby | balsa glider |
| Bobby | toy soldiers |
| Davey | hula hoop |
| Davey | toy soldiers |

### NATURAL JOIN INNER JOIN

Identify matching column names, Column we’re joining by has the same name in both tables

**Result**

|  |  |
| --- | --- |
| **boy** | **Toy** |
| Beaver | balsa Glider |
| Bobby | hula hoop |
| Davey | balsa glider |

## SUBQUERY

A **subquery** is a query that is wrapped within another query. It’s also called INNER query.

OUTER QUERY + INNER QUERY = Query with a subquery

**OUTER QUERY**

**SELECT mc.firstname, mc.last\_name, cj.title**

**FROM current\_job AS cj**

**NATURAL JOIN my\_contacts as mc**

**WHERE cj.title IN (*SELECT title from job\_l istings*);**

***INNER QUERY***

SELECT mc.first\_name, mc.last\_ame, cj.salary

FROM my\_contact as mc

NATURAL JOIN current\_job as cj

WHERE **cj.salary = (SELECT MAX(cj.salary) FROM current\_job cj)**

If a **subquery is used as column expression** in a SELECT statement, it can only **return one** value from one column.

Therefore,

**SELECT mc.first\_name, mc.last\_name,**

**(SELECT state FROM zip\_code WHERE mc.zip\_code = zip\_code) AS state**

**FROM my\_contacts mc**

### NON-RELATED SUBQUERY

If the subquery stands alone and doesn’t reference anything from the outer query, it is a noncorrelated subquery.

**OUTER QUERY**

**SELECT mc.first\_name, mc.last\_name, cj.salary**

**FROM my\_contacts AS mc**

**NATURAL JOIN current\_job AS cj**

**WHERE cj.salary >**

**(SELECT cj.salary FROM my\_contacts AS mc**

**NATURAL JOIN current\_job AS cj**

**WHERE email = ‘andy@weathorma.com’)**

***INNER QUERY***

A non-correlated subquery uses **IN or NOT IN** to test if the values returned in the subquery are **members of a set (or not)**

SELECT mc.first\_name, mc.last\_name, cj.salary

FROM my\_contacts AS mc

NATURAL JOIN current\_job AS cj

**WHERE cj.title IN (SELECT title FROM job\_listings)**

SELECT mc.first\_name, mc.last\_name, cj.salary

FROM my\_contacts AS mc

NATURAL JOIN current\_job AS cj

**WHERE cj.title NOT IN (SELECT title FROM job\_listings)**

### CORRELATED SUBQUERY

The subquery depends on the outer query. It uses the same alias or correlation name that was created in outer query.

SELECT mc.first\_name, mc.last\_name FROM my\_contacts AS mc

**WHERE 3 = (SELECT COUNT(\*) FROM contact\_interest WHERE contact\_id = mc.contact\_id)**

### EXISTS

SELECT mc.first\_name firstname, mc.Last\_name lastname, mc.email email

FROM my\_contacts mc

**WHERE EXISTS**

**(SELECT \* FROM contact\_interest ci WHERE mc.contact\_id = ci.contact\_id)**

### NOT EXISTS

SELECT mc.first\_name firstname, mc.Last\_name lastname, mc.email email

FROM my\_contacts mc

**WHERE NOT EXISTS**

**(SELECT \* FROM current\_job cj WHERE mc.contact\_id = cj.contact\_id)**

## OUTER JOIN

### LEFT OUTER JOIN

The **left outer join** matches every row in the left row in the left table with a row from the right table.

**Table Toys**

|  |  |
| --- | --- |
| **toy\_id** | **toy** |
| 1 | hula hoop |
| 2 | balsa glider |
| 3 | toy soldiers |

**Table Boys**

|  |  |  |
| --- | --- | --- |
| **boy\_id** | **boy** | **toy\_id** |
| 1 | Davey | 2 |
| 2 | Bobby | 1 |
| 3 | Beaver | 2 |

**Result**

|  |  |
| --- | --- |
| **boy** | **toy** |
| Beaver | balsa Glider |
| Davey | balsa glider |
| Bobby | hula hoop |
| NULL | toy soldiers |

**SELECT b.boy, t.toy**

**FROM toys t**

**LEFT OUTER JOIN boys b**

**ON b.toy\_id = t.toy\_id**

So, the left outer join takes all the rows in the left table (the boys table) and matches them to rows in the RIGHT table (the toys table)

A **NULL** value in the results of a left outer join means that the right table has no values that correspond to the left table.

### RIGHT OUTER JOIN

The right outer join evaluates the right table against the left table

|  |  |
| --- | --- |
| **boy** | **Toy** |
| Beaver | balsa Glider |
| Bobby | hula hoop |
| Davey | balsa glider |

**SELECT b.boy, t.toy**

**FROM toys t**

**RIGHT OUTER JOIN boys b**

**ON b.toy\_id = t.toy\_id**

## SELF-REFERENCING FOREIGN KEY

A **self-referencing foreign key** is the primary key of a table used in that same table for another purpose.

## SELF JOIN

The **self-join** allows us to query a single table as though there were two tables with exactly the same information in them.

Boss Table

|  |  |  |
| --- | --- | --- |
| **id** | **name** | **boss\_id** |
| 1 | Elsie | 3 |
| 2 | Snuggles | 1 |
| 3 | Mr. Hobo | 1 |
| 4 | Clarabelle | 4 |

**SELECT c1.name, c2.name AS boss**

**FROM boss\_table c1**

**INNER JOIN boss\_table c2**

**ON c1.boss\_id = c2.id**

**RESULT: Clarabelle**

## UNION

SELECT title FROM job\_current

**UNION**

SELECT title FROM job\_desired

**UNION**

SELECT title FROM job\_listings

Union can only take one ORDER BY at the end of the statement. This because UNION concatenates and groups the results from the multiple SELECT statements

Union Rules

**Rule 1:** The number of columns in each select statement must match.

**Rule 2:** It must also have the same expression ang aggregate function sin each select statement

**Rule 3:** Statements can be any order and it suppresses duplicate values from the results

**Rule 4:** The data type in the column need to either be the same or be convertible to each other

**Rule 5:** If we see duplicate data then we can use **UNION ALL**

### UNION ALL

SELECT title FROM job\_current

**UNION ALL**

SELECT title FROM job\_desired ORDER BY title

Removes any duplicate data

### CREATE TABLE UNION

**CREATE TABLE my\_union AS**

**SELECT title from job\_current UNION**

**SELECT title from job\_desired UNION**

**SELECT title from job\_listings**

## INTERSECT

CREATE title FROM job\_current

**INTERSECT**

SELECT title FROM job\_desired

Show the data that are common between these tables

## EXCEPT

SELECT title FROM job\_current

**EXCEPT**

SELECT title FROM job\_desired

Show all the data from job\_current except that are common with job\_desired

## CHECK

A **check** constraint restricts what values we can insert into a column. It uses the same conditionals as a WHERE clause

CREATE TABLE piggy\_bank(

id INT AUTO\_INCREMENT NOT NULL PRIMARY KEY,

**coin CHAR(1) CHECK (coin IN(‘P’,’N’,’D’,’Q’))**

)

### ALTER TABLE FOR CONSTRAINT

**ALTER TABLE my\_contacts**

**ADD CONSTRAINT CHECK gender IN (‘M’,’F’)**

## VIEW

**CREATE VIEW web\_designers AS**

SELECT mc.first\_name, mc.last\_name, mc.phone, mc.email

FROM my\_contacts mc

NATURAL JOIN job\_desired jd

WHERE jd.title = ‘Web Designer’

A view is basically a table that only exists when we use the view in a query. It’s considered a virtual table because it acts like table, and the same operations that can be performed on a table can be performed on a view

1. We can keep changes to our database structure from breaking applications that depend on our tables
2. View makes our life easier by simplifying our complex query into a simple command
3. We can create views that hide information that isn’t needed by the user

### CHECK OPTION

It checks each query we try to **INSERT** or **UPDATE** to see if it’s allowed according to the **WHERE** clause in our **view table**. In MySQL, we can imitate a **CHECK CONSTRAINT using a CHECK OPTION**

An **updatable view** includes all the **NOT NULL** columns from the tables it references. A non-updatable view is a view that doesn’t include all the not null columns.

**CHECK** constraints and view both help maintain control when we have multiple users

### DROP VIEW

**DROP VIEW**

Drop the view that has been created

## TRANSACTION

A **transaction** is a set of SQL statements that accomplish a single unit of work. During a transaction, all of the SQL statement should be completed without any interference or else none of them should be completed

### ACID

**ATOMICITY**

All of the pieces of transaction must be completed, or none of them will be completed.

Scenario: Power Outage

**CONSISTENCY**

A complete transaction leaves the database in a consistent state at the end of the transaction.

Scenario: Money transferred to savings and then translated into cash

**ISOLATION**

Every transaction has a consistent view of the database regardless of other transaction taking place at the same time.

Scenario: Two person of the same account holder going to the booth at once, only person should be able to retrieve the money, the other have to wait.

**DURABILITY**

After the transaction, the database needs to save the data correctly and protect it from power outages or other threats.

Scenario: Power Outage 🡪 Her records saved to a different location than the main database (backup)

Small Note

**[ALTER TABLE table\_name ADD INDEX (last\_name] quick way to index**

**START TRANSACTION**

It keeps track of all the SQL that follow until we enter **COMMIT or ROLLBACK**

**COMMIT**

If we have got all our statements in place and everything looks good, we **COMMIT** to make it permanent

**ROLLBACK**

If something isn’t quite right, **ROLLBACK** reverse everything to the way it was before we typed **START TRANSACTION**

To make the transaction work with MySQL we need to use the correct **storage engine**

CREATE TABLE(… …)**ENGINE=MyISAM DEFAULT CHARSET=latin1**, this cannot be used.

We need to make sure that our storage engine is either **BDB or InnoDB**, there are two choices that support transactions.

**ALTER TABLE** table\_name **TYPE = InnoDB**

**START TRANSACTION**

SELECT \* FROM piggy\_bank

UPDATE piggy\_bank set coin = ‘Q’ WHERE coin = ‘P’

SELECT \* FROM piggy\_bank

**ROLLBACK**

SELECT \* FROM piggy\_bank

**START TRANSACTION**

SELECT \* FROM piggy\_bank

UPDATE piggy\_bank set coin = ‘Q’ WHERE coin = ‘P’

SELECT \* FROM piggy\_bank

**COMMIT**

SELECT \* FROM piggy\_bank

## PASSWORD PROTECT

**SET PASSWORD FOR ‘root@localhost’ = PASSWORD (‘rootpass’)**

## CREATE USER

**CREATE USER jack IDENTIFIED BY ‘jackinthebox’**

jackinthebox, over here is the password

## GRANT

We can control exactly what user can do to tables and columns with the **GRANT** statement

Giving permission to the user

**GRANT SELECT ON table\_name TO jack**

## REVOKE

Remove the privilege from the table

**REVOKE SELECT ON table\_name FROM jack**

### GRANT OPTION

**REVOKE GRANT OPTION ON**

**DELETE ON table\_name**

**FROM jack, jill**

Jack and Jill can delete things from the chores table, but they can’t give anyone else the delete privilege to anyone else.

Admin🡪 gives **DELETE** privilege 🡪 Jack

Jack 🡪 gives **DELETE** privilege 🡪 Jill

Admin 🡪 removes **DELETE** privilege from Jack

Therefore, Jill will also lose the **DELETE** privilege

### CASCADE

REVOKE DELETE ON table\_name FROM Jack **CASCADE**

Admin🡪 gives **DELETE** privilege 🡪 Jack

Jack 🡪 gives **DELETE** privilege 🡪 Jill

Admin 🡪 removes **DELETE** privilege from Jack

Therefore, Jill will also lose the **DELETE** privilege

### RESTRICT

REVOKE DELETE ON table\_name FROM Jack **RESTRICT**

Admin🡪 gives **DELETE** privilege 🡪 Jack

Jack 🡪 gives **DELETE** privilege 🡪 Jill

Admin 🡪 removes **DELETE** privilege from Jack

Admin will receive an error

## CREATE ROLE

**CREATE ROLE data\_entry**

**GRANT SELECT, SELECT ON table\_name TO data\_entry**

**GRANT data\_entry TO table\_name**

GRANT data\_entry TO table\_name **WITH ADMIN OPTION**

REVOKE data\_entry FROM table\_name **CASCADE**

REVOKE data\_entry FROM table\_name **RESTRICT**

## DROP ROLE

**DROP ROLE data\_entry**

## COMBINING CREATE USER AND GRANT

**GRANT SELECT ON table\_name**

**TO user\_name IDENTIFIED by ‘password’**

## ALL

Greater than all finds any values larger than the biggest value in the set

SELECT name, rating FROM restaurant\_ratings WHERE **RATING > ALL**

(SELECT rating FROM restaurant\_ratings WHERE rating >3 and rating <9)

Less than all finds any values smaller than the smallest value in the set

SELECT name, rating FROM restaurant\_ratings WHERE **RATING** **< ALL**

(SELECT rating FROM restaurant\_ratings WHERE rating >3 and rating <9)

## ANY

Greater than any finds any values larger than the smallest value in the set

SELECT name, rating FROM restaurant\_ratings WHERE **RATING > ALL**

(SELECT rating FROM restaurant\_ratings WHERE rating >3 and rating <9)

Less than any finds any values smaller than the largest values in the set

SELECT name, rating FROM restaurant\_ratings WHERE **RATING** **< ALL**

(SELECT rating FROM restaurant\_ratings WHERE rating >3 and rating <9)

## SOME

It works same as any

## BOOLEAN

We can true or false

## INT

|  |
| --- |
| **DATA TYPES** |
| TINYINT |
| SMALLINT |
| MEDIUMINT |
| BIGINT |

## DATE AND TIME

|  |  |
| --- | --- |
| **DATA TYPES** | **FORMAT** |
| DATE | YYYY-MM-DD |
| DATETIME | YYYY-MM-DD HH:MM:SS |
| TIMESTAMP | YYYYMMDDHHMMSS |
| TIME | HH:MM:SS |

SELECT **DATE\_FORMAT(column\_name, ‘%M Y’)** FROM table\_name

## TEMPORARY TABLE

This table just for the session, it deletes out automatically

**CREATE TEMPORARY TABLE** table\_name(

column\_name INT,

column\_name VARCHAR(40)

)

Shortcut

**CREATE TEMPORARY TABLE table\_name AS SELECT \* FROM table\_name**

## CASTING

|  |
| --- |
| **DATA TYPE** |
| CHAR() |
| DATE |
| DATETIME |
| DECIMAL |
| SIGNED [INTEGER] |
| TIME |
| UNSIGNED [INTEGER] |

**SELECT CAST (‘2005-01-01’ AS DATE)**

**SELECT CAST (2 AS DECIMAL)**

SELECT **CAST(column\_name, TYPE)** FROM table\_name

We **can’t use** cast, for **decimal to integer** conversion

## WHO AM I

**SELECT CURRENT\_USER**

## MATH FUNCTION

|  |  |  |  |
| --- | --- | --- | --- |
| **ABS(x)** | Returns the absolute | **RADIANS(x)** | Degrees to radians |
| **ACOS(x)** | Returns the arccosine | **RAND(x)** | Random floating value |
| **ASIN(x)** | Returns the arcsine | **ROUND(x)** | 1.23 → 1, 1.56 → 2 |
| **ATAN(x,y)** | Returns arctangent | **ROUND(x,y)** | y represents decimal places |
| **CEIL(x)** | 2.34 🡪 3 | **SIGN(x)** | Returns 1 when x positive |
| **COS(x)** | Return the cosine (rad) | **SIN(x)** | Sinusoidal |
| **EXP(x)** | Returns the exponential | **SQRT(x)** | Square root |
| **FLOOR(x)** | 1.32 🡪 1 | **TAN(x)** | Tangent |
| **LN(x)** | Natural logarithm | **TRUNCATE(x,y)** | Cuts down the value |
| **LOG(x)** | Natural log with base 10 | **COT(x)** | Returns the cotangent |
| **LOG(x,y)** | Natural log with base y | **FORMAT(x,y)** | 3,451,000.50 |
| **MOD(x,y)** | Remainder of division |  |  |
| **PI()** | 3.142 |  |  |
| **POW(x,y)** | 3^2 = 9 |  |  |