Create a new database

```
CREATE DATABASE <databaseName> DEFAULT CHARACTER SET utf8;
```

Create a table:

```
CREATE TABLE <tableName>(
    <field1Name> <field1Type>(<maxSize>),
    <field2Name> <field2Type>(<maxSize>),
    .
    .
    .
);
```

Insert elements:

```
INSERT INTO <tableName> (<field1Name>,<field2Name>,....) VALUES (<data1>,
<data2>,.....)
--inserts data into tableName, data which is not mentioned is marked as
"NULL"
```

Read elements:

```
SELECT <field1Name>,<field2Name>,... FROM <tableName> WHERE <condition,
e.g. <field1Name>=<someData>>
--returns only the mentioned fields
```

```
SELECT * from <tableName> where <condition>
-- returns all fields where condition is true
```

```
SELECT * FROM <tableName> ORDER BY <someFieldName>
--list is sorted acc to someFieldName
```

```
SELECT COUNT(*) FROM <tableName> where <condition>
--gives number of rows that would've been returned
```

Possible conditions:

- field=data
- field like '%key%': means something like ajshdlakeylahdoai.
- 1: means every entry
- can be anded/ored.
  etc.

Delete elements:

```
DELETE FROM <tableName> WHERE <condition similar to select>
```

Update elements:

```
UPDATE <tableName> SET <someFieldName>=<data> WHERE <condition>
```

Data Types:

1. String:
    1. `CHAR`- char array of maxSize, efficient if size is always near to maxSize
    2. `VARCHAR` - char array of variable size, but not greater than maxSize
2. Text fields: not good for indexing/sorting
    1. `TINYTEXT` - upto 255 chars
    2. `TEXT` - upto 65K
    3. `MEDIUMTEXT` - upto 16M
    4. `LONGTEXT` - upto 4G
3. Numbers:
    1. `TINYINT` : 8bit
    2. `SMALLINT` : 16bit
    3. `INT` : 32bit
    4. `BIGINT`: 64bit
    5. `FLOAT`: 32bit float
    6. `DOUBLE`: 64bit float
    - can be unsigned
4. Dates:
    1. `TIMESTAMP` : 'YYYY-MM-DD HH:MM:SS' after 1970
    2. `DATETIME` : 'YYYY-MM-DD HH:MM:SS'
    3. `DATE` : 'YYYY-MM-DD'

4. `TIME` 'HH:MM:SS'

- `AUTO_INCREMENT` : if not mentioned while creating, is set as previous entry + 1
- `PRIMARY_KEY(fieldName)` : a way to access a specific row of a table is using that fieldName. Uses HashMap-like technique
- `INDEX(fieldName)` : makes 'where' statements kinda more efficient I guess, uses binary tree to store, so it is easier to look up.
- Foreign key: used to "reference" a row of another table from this table. Value of foreign key for this table is usually the primary key for another table.

```
CREATE TABLE <name>(
    <field1Name> <field1Type>(<maxSize>) NOT NULL AUTO_INCREMENT,
    <field2Name> <field2Type>(<maxSize>),
    .
    .

    PRIMARY KEY (<field1Name>),
    INDEX using BTREE (<field2Name>),
    CONSTRAINT FOREIGN KEY (<fieldName>) REFERENCES <anotherTableName>
(<fieldName>)
    ON DELETE CASCADE
    --Means if entry from another table is deleted, all the entries in this
table that reference those entries will be deleted.
    ON DELETE RESTRICT
    --Don't allow deletion of entries from another table that are
referenced by this table
    ON DELETE SET NULL
    --set the values of foreign keys as NULL if the entry referenced is
deleted.
    ON UPDATE, -- same 3 options as ON DELETE
);
```

While reading entries from database, if entries are from multiple tables, `JOIN` keyword is required.

```
SELECT table1.field1, table1.field2, table2.field1.... FROM table1 JOIN
table2 JOIN table3.... ON table1.someField = table2.someField AND/OR/..
--JOIN keyword forms all possible PnCs of combined rows, that is
n1*n2*n3... where ni is no. of rows in tablei. ON is like an if statement
which says return only those rows where the condition is satisfied.
```

Many to many relationships, e.g. accounts and courses. Multiple accounts can be taking a single course, and multiple courses can be takn by a single account. So this is a many to many relationship.
To model this, 1 intermediate table is used.
e.g. 1 table contains account details with account ID another table contains course details with course ID
And a third table contains: account_ID | course_ID | role(teacher/student)

account_ID and course_ID are both foreign keys, which refer a row in account table and course table respectively.

```
--if account name, course, role were to be fetched for all courses,
SELECT account.name, course.title, joiner.role FROM account JOIN course
JOIN joiner ON account.account_ID=joiner.account_ID AND
course.course_ID=joiner.course_ID;
```