# Relational Database Model

1. DBMS’s goal is to provide
   1. Data integrity / consistency
   2. Concurrent access
   3. Efficient storage and access
   4. Standardized format / administration
   5. Standardized query interface (sql)
2. Relation is a formally as a set of cartesian products of attributes
3. Tables are explicitly constructed relation, whereas relations are abstractions (including views or temporary table)
4. Changing column order changes the relation, however, changing tuple order does not change the relation b/c relations are sets
5. Relations are defined by schemas, which are again, denoted as: table name(col1: dtype col1, …)
6. Joining the various relation schemas is how you define a data model
7. Compound key - multiple columns that identify unique rows
8. Foreign key, a primary key in one relation can be a column in another, so you can join different relations on the primary key
9. Normalization: data is normalized if it is not redundantly stored, each record is stored exactly one

# SQL

Three type of programming languages:

1. proceredual : C, Java, Python, (follow steps)
2. Functional: Scala, Lisp, ML etc. (nested functions)
3. Declarative: SQL, do what i tell you to do, computer optimizes the code in terms of relational algebra (think of it more as a protocol rather than language)

## Select operator:

* Grabs data you want, filter using a where clause
* Can subset columns you want by selecting the columns you want

## 

## Joins

* Allows users to connect information in different tables by joining ona common key
  + Cross join - cartesian product
  + Left join - takes data in table a and adds on matching data from table b
  + Right join takes data in table b and adds on matching data from table a
  + Full outer join if theres any match, join table a and b, if no match, just report data from the individual tables
  + Inner join - intersection

## Modifying Data:

1. Insert into table(col1 .. col n) values ( val 1 … val n), (val1.. Valn n) would insert two rows
2. Update table set column1 = value 1, column2 = value 2 where ..
3. Sql injections, if you pass a column into a sql command, and the column takes a value that is a cleverly instructed sql command, the sql compiler will run the sql command that is supposed to be a column name. This is why if you name your son drop table, when querying the name column, you could drop the whole table.
4. What you should do to sanitize your code is pass in a variable that will take a non-executable string value, such that any sql commands in a column name are actually not executed

## Aggregating Data:

* Group by syntax, with avg(), sum(), count(), count(\*) includes nulls, min, max, etc. etc. as being aggregation functions
* Having is a filter after the group by
* Where filters go before the group by

## Indexing

* If you group things typically together, you want to have memory in disk of things next to each other that will be grouped together
  + I.e. if you group by zip often, rather than having zip randomly assigned in memory, you would want sequential zips next to each other in memory
* If we have a subset of our data set that we use often, i.e. country = ‘us’ indexing allows us to cache that subset onto memory so we can access it faster (just the index)
* Cons - take time and space, especially with an index spanning multiple columns
* Updates become slower, because you have to re-index on the updated table
* No guarantee that they will help in all queries
* When to index:
  + When data is read more often than written
  + When queries are predictable
  + When queries rely on a small number of attributes
  + Best practice to have indexes with a low number of rows associated with them

# Transactions

## Managing concurrent users:

Example, a file system visualized in a graph:

* Constraints on our FS:
  + Graph must be connected
  + Vertices stored in nodes.dat
  + Edges stored in edges.dat
* Challenge, add a new node and two edges:
  + Can’t add vertex first to the graph, as the graph won’t be connected
  + Can’t add two edges to the graph, as there is no node to connect those edges to each other
* What happens if someone wants to Read /Write the data and then another user wants to read from that data at the same time:
  + Bigger problem, multiple users reading + writing from the same data at the same time

## ACID principles of database management:

* A: Atomicity
  + The concept that a sequence of actions either must entirely happen or will not happen at all, managed by transactions
* C: Consistency
  + Transactions move from one valid state to another
  + Cannot leave the database in an invalid state
* I: Isolation
  + Concurrent operations do not depend on order of execution
* D: Durability
  + Completed transactions are permanent (flushing to disk before completion)

In practice, this means:

* When modifying tables we can use the following syntax:
  + Begin Transaction; queries; commit; or
  + Begin Transaction; queries; rollback;
  + Alternative syntax might be START transaction
* Analogous to a try, except clause in python
* Consistency is maintained by schema, atomicity
* If a query fails mid-transaction, uncommitted changes will be abandoned
  + If data does not fit the schema, the operation fails immediately, and it goes back to the last valid state
  + Cascading delete can be used to maintain foreign key constraints
* Isolation
  + Locks the database during modification
  + Only one thing at a time can be modify the data
  + Bad for when multiple users are trying to modify the database, especially on distributed databases
  + Map Reduce side steps Isolation, because you can’t re-write existing data, but you write new data which doesn’t require locking

The lab using github exemplified all of these practices