9/16/2019 Lecture 2

Recap diagram from last time. (Slide 1)

Going to start with how to represent data.

Tabular data is the norm -- could be CSVs, spreadsheets, databases or dataframes

Today we're going to talk about tabular data representations ("relations") and operations over them ("relational algebra" + SQL)

Basic tabular representation
- typed records
- ordered / unordered?
- nested?

Example:

bandfan.com

members
- id
- names
- birdthdays
- addresses
- emails

sam, 1/1/2000, 32 vassar st, srmadden
tim, 1/2/1990, 46 pumpkin st, timk
...

bands
- id
- name
- genre
- ...

We call the names and types of fields in a table the "schema"

Key challenge:

data needs to capture *relationships* between multiple data sets

examples:

members are fans of bands
bands play in shows
...
employees in departments working on projects
musicians in bands signed with labels
students in classes in universities
cars made by manufacturers bought by customers
parents with children who attend school
patients of doctors in different hospitals
...

how to represent these relationships?
why is this complicated?

Different types of relationships:

One to many: each member is a fan of many bands
Many to many: each band plays multiple shows, multiple bands can play at a show

How to represent this:

Try 1 (Show slide)

Member-band-fans

What's wrong with this representation?
Duplicate info - why is that bad?
Inconsistency
Wasted space
No ability to represent missing data
Add NULL?

Try 2:

Still redundant information

Try 3:

Eliminates redundancy

This is a general approach: for many to many relationships, create a relationship table to eliminate redundancy

Generally works but can get complicated when you start adding complex restrictions; for example, suppose we wanted to allow each member to be a fan of just one band per genre?

It's not possible to represent this in a single table without duplicating information, or requiring me to connect several tables together to do it

What about one to many relationships? Show slide -- can add a reference column to the original table

How to devise a schema? Most common way is to write down the nature of the relationships (one to many, many to one), as well as the attributes, and then the tables that represent it. Sometimes people use what's called an *entity relationship diagram.*

This is 95% of what you need to know about database theory….

Study break

Part II - Operations on Relations

We're going to study lots of different ways to manipulate tables -- and of course it's possible to perform arbitrary transformations over them with programs.

Suppose we just want to focus on the problem of extracting a set of records of interest from a collection of tables.

We need to find a way to extract columns and rows of interest, and a way to follow paths from one table to another. A fancy name for this is a *relational algebra.*

Here, a *relation* is just a table with a schema, with unordered rows and no duplicates

Algebra just refers to the fact that we have set of operations over relations that is *closed*, i.e., each operation on a relation (or pair of relations) produces another relation.

Call a collection of relations a "database"

Main operations:

> Projection (π(T,c1, …, cn)) -- select a subset of columns c1 .. cn
> Selection (sel(T, pred)) -- select a subset of rows that satisfy pred
> Cross Product (T1 x T2) -- combine two tables
> Join (T1, T2, pred) = sel(T1 x T2, pred)

Example showing how join & select works -- find creed shows

Plus various set operations (UNION, DIFFERENCE, etc)

Notice that basic ops are all set oriented -- i.e., they produce another valid relation

Although we won't go into it much, one of the cool properties of these operations is that they obey interesting algebraic identities that allow a system that executes relational algebra expressions to choose the order in which it does work, for example:

<u>sel reordering</u>
> Sel1(Sel2(A)) = Sel2(Sel1(A))

<u>sel push down</u>
> Sel(A join B, pred) = Sel(A, pred) join Sel(b, pred)

Find the dates of Creed shows

proj (join (sel(bands,name="creed"), shows, shows.bandid = bands.id), shows.date)

Show data flow diagram slide

This suggests a natural implementation -- we aren't going to talk much about implementations of the low level operators or executors -- although we will revisit a bit later, but it's good to have a mental model of this.

Ex 2: Find the bands tim likes

Mbf = Member-band-fans

join(join(sel(fans, name='tim'), mbf, mbf.fanid = fans.id), bands, bands.id = mbf.bandid)

SQL -- most popular physical embodiment of relational algebra

Show a few example SQL queries (see SQL querie)

Note that SQL is "Declarative" - we say what we want, not how to achieve it
Even for a simple selection, may be:

1) Iterating over the rows
2) Keeping table sorted by primary key and do binary search
3) Keep the data in some kind of a tree structure and do logarithmic search

Note that as a user of a SQL database, you don't need to know how the system is evaluating the query, or even what the physical representation of the data is.

SQL provides "physical data independence" -- of course there is some underlying representation of the data, but no matter the representation, the same SQL queries will still run over it.

This can be both a blessing and a curse -- cool because as a user you don't have to worry about it, but bad because it can make understanding bad performance hard.

Show some examples of indexes / plans:

```
SELECT fans.name
FROM bands
JOIN band_likes bl ON bl.bandid = bands.id
JOIN fans ON fans.id = bl.fanid
WHERE bands.name = 'Justin Bieber'
```

Look at physical plan chosen
Note effect of creating an index on bands.name
For small bands table, has no effect
For larger table, will choose to use index
Depends on clustering