CSC343 Notes

Introduction to Databases

https://github.com/ICPRplshelp/

Last updated January 16, 2023

1 What is Data?

- Bits that represent values, such as:
  - Numbers
  - Strings
  - Images
- What do we want to do with it?
  - Manage it!!!
- We can manage a large collection of data with files
  - We used flat files, K/V pairs
  - If you combine K/V pairs with your favorite programming language, you get a database.
  - Of course, you'll have to implement all its methods yourself.
- The first commercial databased evolved using plain text. Now, how do we deal with:
  - Duplication
  - Organization (it is incredibly hard, especially when you have more than one person)
  - You'll have to reinvent various wheels:
    * Search
    * Sort
    * Modify, and so on
  - None of these would be optimized

What do we do instead

- We want an efficient, optimized system that specializes in handling **inconvincibly large amounts of data.**

In fact, some of the largest databases would already cram your hard drive in a second/

1.1 Databases and DBMS (Database Management System)

A **DBMS** is a powerful tool that can:

- Manage large amounts of data
- Allow it to persist for long amounts of time (protect it over time from being overwritten)

Every DBMS has a data model. It describes

- Structure (lowest level: rows and columns)
- Constraints (range, types)
- Operations (find all students with grades ≥ 85)

We'll be looking with the **relational data model.** This has been very widespread since it has been invented. It has been proposed as an efficient model.

1.2 The Relational Data Model

CSVs. Spreadsheets. They look a lot like them. The main concept is a relation. The concept of relation is borrowed from math.

A relation is a **SET** of **TUPLES.** Here's an example:

$$\{(x,y) \in \mathbb{R}^2 \ : x < y\}$$

That defines a relation. Every pair of values that obeys $x < y$ will be in it.

- A **column** (attribute, field) goes vertically.

- A **row** (tuple) goes horizontally).
- A table is a set of tuples (rows) (what about the header?)

A **schema** is a namespace where we collect some relations. When we do a lot of work with databases, we are likely to have some completely different areas of interest, and this helps us organize them at the top level.

1.3 What does a DBMS provide

- Can **explicitly** specify the logical structure of the data
  - And **enforce** it
- Can query (ask questions and it will respond) or modify data
  - Best to keep query and modify separate
- Good performance under heavy loads
  - We're talking about billions of data. We're getting into the range of 10^{15} but we're starting to push way more
- Durability of the data
  - Keep it safe and intact (data integrity)
- Concurrent access by multiple users
  - MERGE CONFLICTS!!! Us or us and our partners? We need to have ways of having concurrent access without causing a mess.
  - Suppose tables A and B are bank accounts. We have a couple of queries that remove $100 from A and deposit that amount in B. What happens if another user checks A before the $100 is removed and B after $100 is deposited?
  - If you're developing databases for multiple concurrent users, you'll have to worry about this exact issue, and it's important to pay attention to it.

The architecture of a DBMS relies **heavily** on the underlying operating system. You need access to direct pieces of the operating system. The DBMS sits between users and the data itself. Sometimes, it sits between an app you write and the database.

Rather than allowing someone who is pointing and clicking on the webpage, we have a DBMS that ensures that the queries sent by the person are well-formed and efficient.

Or maybe a Python program is forming proper DBMS queries first. We rarely have an end user directly interacting with the database.

1.4 Relations

The cartesian product is every possible ordered tuple.

- A domain is a set of values
- Suppose $D_1, D_2, \ldots, D_n$ are domains
- $D_1 \times D_2 \times \cdots$
  - Is the set of all tuples $(d_1, d_2, \ldots, d_n)$
  - Every combination of a value from D_1, D_2, and so on

1.4.1 Example of a Mathematical Relation

Let $A = \{p, q, r, s\}, B = \{1, 2, 3\}, C = \{100, 200\}$

Anything that is a subset of $A \times B \times C$ is a **relation** on A, B, C.

For example, $R = \{(q, 2, 100), (s, 3, 200), (p, 1, 200)\}$ is a relation on A, B, C.

Database tables are relations. The order of rows does not matter. We can represent a table as:

$$\{\text{row 1 info}, \text{row 2 info}, \ldots\}$$

1.5 Relation Schemas vs. Instances

- Schema is the **definition of the structure** (constraints, restrictions)
  - A database schema is a set of relation schemas
- We have notations for expressing a relation's schema:
  `Teams(Name, Homefield, Coach)`
- An instance is a particular data in a relation. It is a snapshot of a database at a point of time.
  - Instances change **constantly,** schemas change **rarely** and are inconvenient to change.
  - A database instance is a set of relation instances
  - As soon as we put in any data into a database schema, we get an instance.
- Conventional databases store the **current version** of the data. Databases that record history are temporal databases (diff files).

1.6 Terminology

- Relation (table)
- Attribute (columns) or fields
- Tuple (row)
- Arity (no. of attributes / column count)
- Cardinality (no. of tuples, or rows; size of R. Usually finite)

Relations are sets; no duplicates allowed, and we don't care about the order of the tuples.

There is another model called a **bag** – sets that allow duplicates / multisets. Commercial DBMSs use this model, but for now we'll stick with relations as sets.

- PostgreSQL uses bags
- Apparently they are more efficient

1.7 Making Constraints

TL: DR use IDs to prevent duplicates. Until we want to do some domain or range restrictions, but that's something else.

- We have `Teams(name, homefield, coach)` and `Games(hometeam, awayteam, homegoals, awaygoals)`.
- Do we allow duplicates? Not up to us. The responsibility for that decision is not you, the domain expert (a.k.a. you'll be asking them. Don't make the decisions, do what you're ordered to).
- Suppose we want to allow duplicate names and multiple teams with the same home field.
  - The schema allows it
- The only thing that would distinguish the two teams apart is another attribute where duplicates are not allowed.
- **What if we don't want that?** We can **constrain the data.**

A constraint that forbids duplicates:

$$\nexists \text{ tuples } t_1, t_2 \text{ such that } (t_1.name = t_2.name) \\ \wedge (t_1.homefield = t_2.homefield)$$

THEN, if we know the values for $(name, homefield)$ then we can look up any team we want.

SUPERKEY – a set of 1 or more attributes whose combined values are unique. No two tuples can have the same values on all these attributes.

1.7.1 With Courses

For example: `Course(dept, number, name, breadth)`

- If `<"csc","343","Intro Databases",True>` were an instance

- `<"csc", "343", "AAAAAAAAA", True>` violates `{dept, number}` being a superkey.

> If `{dept, number}` is a super key, then so is `{dept, number, name}`.

But we are more interested in a MINIMAL set of attributes with the superkey property.

- Minimal means it's no longer possible to remove attributes from the superkey without making it no longer a superkey

For example, if

- `{st. number , utorID}` is a superkey
- `{st. number, utorID, wordle average, fav color}` is a super key
  - I can remove `wordle average` and `fav color` from that key and it would remain a superkey.

1.8 Key

A **key** is a minimal superkey. By convention, we underline a key. Course (dept, number, name, breadth)

The word superkey comes from the word "superset". This means that somewhere, a superkey must contain the **key** as a subset.

1.9 The Importance of IDs

Not everything can avoid duplicates, so sometimes we introduce attributes. This ensures that all tuples are unique. (Integrity constraint)

- USE IDs!!!
- Every site does this (unless there's something wrong with the person making the site)

1.10 References between Relations

Better to use separate tables for separate concepts. Rather than repeat information already stored elsewhere, we store the key instead of all the data associated with the thing.

- For example: for an artist, put their ID instead of literally everything else (if the ID is sufficient to identify the artist).

1.11 Foreign Keys and Constraints

If in one table we have an attribute that is the key to another table, that is called a foreign key. Firstly, notation:

Notation: $R[A]$ is the set of all tuples from R only with the attributes in A.

We can declare foreign key constraints:

$$R_1[X] \subseteq R_2[Y]$$

If attribute X is a foreign key in relation 1, all values of X must occur in values of attribute Y in relation 2.

- Y must be a key in R_2.

To write that a bit more clearly:

- For attribute X, if we observe in the relation R_1
- If we perform `set( the column of attribute X )`
- That must be a subset of `set(the column of attribute Y)`
  - Most likely, X and Y share the same attribute name.

2 Relational Algebra

> Source for many of these notes is from HERE.

Queries operate on relations and provide relations as a result.

The simplest query is just the relation's name. If we run that query on the database: `student`, we just get `student`, the database, as the return value.

2.1 Select and Project (R was confusing)

Here are some operators:

- `select` ($\sigma_{\text{condition}}$ Expr): filter the table by getting rid of the rows that don't meet the condition. Use the logical and operator $\wedge$, or the or $\vee$ operator if you want some binary operators.
  - For example, $\sigma_{\text{GPA}>3.7 \wedge \text{HS}<1000,\ \text{major}=CS}$
- `pro ject` ($\pi_{\text{attribute}_1,\text{attribute}_2,\ldots}$ (Expr))
  - Project gets rid of all the attributes not subscripted. It also gets rid of duplicates afterwards, for the purposes of this course (MAYBE not for SQL).
- `Expr` is any expression that returns a relation, which can be passed in. They're like function arguments.

We can combine multiple operators by compositing them:

$$\pi_{\text{sID, sName}} (\sigma_{\text{GPA}>3.7} \text{ Student})$$

2.2 The Cross Product

- The cross product: gluing two relations together, and attributes of each relation will be made unique (e.g. `student.ID`, `apply.ID`).
  - If I run Student $\times$ Apply, if student has s tuple and apply has a tuples, the result of the Cartesian will have $s \cdot a$ tuples.

- You're telling me, for each row in student, I'm making one role for each row in apply? Is there any use for this? Yes, if you combine it with multiple operators.

Given `Student` and `Apply` tuples, I want names and GPAs of students from high school student count over 1000 who applied to CS and were rejected. Here's what I do:

$$\pi_{\text{name,GPA}}\left(\sigma_{\text{student.sID}=\text{apply.sID}\wedge\text{HS}>1000\wedge\text{major}=\text{cs}\wedge\text{dec}=R}\left(\text{Student}\times\text{Apply}\right)\right)$$

The magic of this, is that `student.sID = apply.sID` removes all "garbage" rows created by the cartesian product. Instead of getting a square, we get its diagonal.

2.3 Natural Join $\bowtie$

Performs cross product but enforces equality on all attributes with the same name if the two tables joined have attribute names in common. This prevents the need for composition with $\sigma_{\text{s.something}=\text{a.something}}$.

The operator is $\bowtie$, called a bowtie. Now, we can make a more concise expression:

$$\pi_{\text{sName, GPA}}\left(\sigma_{\text{HS}>1000\wedge\text{major}=\text{cs}\wedge\text{dec}=R}(\text{Student}\bowtie\text{Apply})\right)$$

I'm pretty sure the $\bowtie$ operator is associative, so get creative with joining more than two things at once. By the way:

$$E_1\bowtie E_2\equiv\pi_{\text{Schema}(E_1)\cup\text{Schema}(E_2)}\left(\sigma_{E_1A_1=E_2A_1\wedge E_1A_2=E_2A_2\wedge\cdots}\left(E_1\times E_2\right)\right)$$