CSC343 Notes

Introduction to Databases

https://github.com/ICPRplshelp/

Last updated February 13, 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 with the exception of the order of attributes, so get creative with joining more than two things at once. By the way:

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

2.4 Edge Cases For Natural Join

- No attribute names in common
  - Results in auto $\times$ due to vacuous truth

- Has common attribute names but no matches
  - Results in no tuples

2.5 Renaming

Natural join can over-match: two objects may mean different things but its `str()` returns the same thing. What do we do?

What if they under-match (`.equals(...)` is something we want to define by custom)? Use

$$\bowtie_{\text{condition}}$$

2.6 Set Operations

ATTRIBUTES MUST MATCH EXACTLY FOR BOTH OPERANDS, OTHERWISE YOU'LL GET AN ERROR MESSAGE

Relations are sets, so we can use set operations. If you're doing this on a tuple of relations, they must have the same number of attributes with the same name, and in the same order.

- Union joins all the elements in one set and all elements in the other set, but no repeats.
- For intersections, we only keep the elements in common.
- For difference, it only has elements in the left operand but not the right operand.
  - Not commutative

If the names or order mismatch, we can:

- Rename attributes
- Use π to permute (re-order)

2.7 Renaming

Useful if you want to take a product of a table and itself.

$$\rho_{\text{new\_name}}$$

To prevent the issue of pairing `(1, 2)` and `(2, 1)`, you could use $\sigma_{T1.i1<T2.i1}$ to ensure that the pairs you get are sorted. This prevents pseudo-duplicates, and also prevents duplicates as well.

2.8 Renaming Attributes

You can't mutate an attribute to do this, but you can always create a new one:

$$NewRelation(a1, a2) = \pi_{x, y} OldRelation$$

This redefines its attribute names.

2.9 Max and Min

Where i is the score:

$$S - \underbrace{(\pi_{\text{All of } T1} \sigma_{T1.i<T2.i} (\rho_{T1}(S) \times \rho_{T2}(S)))}_{\text{not max}}$$

Flip the direction of the inequality to get the `not min` set.

2.10 Occurred At Least Twice

Find all people who did this twice. Then oID is an instance of a person doing something, which is part of the distinguish step.

If you want to find the ID of an something that occurred at least twice, where oID is the disambiguator (key):

$$\pi_{T1.ID}\sigma_{\underbrace{T1.ID = T2.ID}_{\text{match}} \wedge \underbrace{T1.oID < T2.oID}_{\text{distinguish}} \wedge \underbrace{T1.A = N \wedge T2.A = N}_{\text{check}}}(\rho_{T1}(S) \times \rho_{T2}(S))$$

We use $<$ to prevent any issues with doubling up, to only get a triangle instead of a rectangle.

You can omit the check component if everything is a pass.

2.11 Occurred Exactly Twice

Use set minus: Occurred at least twice minus Occurred at least three times

$$\sigma_{\substack{\underbrace{T1.ID = T2.ID = T3.ID}_{\text{match}} \wedge \underbrace{T1.oID < T2.oID < T3.oID}_{\text{distinguish}} \\ \wedge \underbrace{T1.A = N \wedge T2.A = N \wedge T3.A = N}_{\text{check}}}} \begin{pmatrix} \rho_{T1}(S) \\ \\ \rho_{T2}(S) \times \rho_{T3}(S) \end{pmatrix} \times$$

Twice minus three times:

$$\text{At Least Twice} - \text{At Least Three Times}$$

2.12 Occurred The Most Amount Of Times

This cannot be done in relational algebra.

2.13 Every

$AllOfThem(ID, Trait)$ is what happened; $RequiredAll(Trait)$ contains attributes that all "people / unique identifiers" need to remain in the query. For example, if I want to query the following: the course codes of all courses offered in 20229 and 20231 (the trait would be the terms), `RequiredAll` would just contain the terms: `20229, 20231`.

$$EveryPossible(ID, Trait) = (\pi_{ID} AllOfThem) \times \underset{\text{traits only}}{RequiredAll}$$

$$Missing(ID) = EveryPossible - AllOfThem$$

$$(\pi_{ID} AllOfThem - \pi_{ID} Missing)$$

In short:

- Make all combos that should've occurred.
- Subtract those that did occur to find those that didn't.
- Subtract what was missing from those that did occur after projecting each to persons only.

Here, $Missing$ is a set of IDs that don't match the condition.

2.14 Set Difference Tips

Set difference takes away everything in the first set that is not in the second set. For example:

$$\{A, B, C, D\} - \{A, X, C, D, F\} = \{B\}$$

You might want to use this to find out the complement of things you would get in a natural join.

Items on the right set have no impact on the result.

2.15 Expressing Constraints

Query a violator and $=$ it to $\varnothing$ to say it can't occur.

Or if we want a query to capture everything, say that selecting and processing does nothing.

$$A \subseteq B \Leftrightarrow A - B = \varnothing$$

3 SQL

RA	SQL
Pi	SELECT
Sigma	WHERE for rows, HAVING for aggregates after a group by
Renaming columns or relations	As
Cartesian product	Comma (,)

SQL is a declarative language. Take this with a grain of salt: say what you want but express it in a very specific form and order to get what you're asking for.

3.1 Aggregate Data and Group By

What does this query do?

```sql
SELECT oid, avg(grade)
FROM took
GROUP BY oid;
```

Gives me the course average for each offering.

We can obtain more idea and give more meaningful names:

```sql
SELECT oid as offering, avg(grade),
min(grade), max(grade)
FROM took
GROUP BY oid;
```

Newlines don't really matter but beware of commas. This also tells us about the order in which queries are made.

1. State your tables.
2. State how you're going to join them.
3. Which rows are you going to look at?
4. If you want to group portions (rows) of the table, you can use GROUP BY
5. Of those groups, which groups should we leave in or keep (HAVING)
    a. You must use `HAVING` right after `GROUP BY`
6. Order what you see (`ORDER BY`): the first argument takes the most priority
7. Tell us which columns we want to have in an output (`SELECT`).

Aggregate functions:

- min, max, avg, and all the tools you have in stats.
- count, which counts the number of rows in a group. If you want to count how many unique items are in a group, use `count(distinct columnName)`. Otherwise, just use `count(*)`
- REMEMBER THAT `distinct` has a low precedence, so `distinct dept, instructor` should be viewed as `distinct (dept, instructor)`.

3.2 Legal and Illegal Queries

Illegal queries will result in an error when sent. The list below may not list everything, so use your common sense.

3.2.1 Ambiguous with Group By

When you use GROUP BY, you may not select something that was not mentioned in `group by` without using `min, max, average`, or so on. Otherwise, you will get `ERROR: column reference "NAME" is ambigious`

You cannot select anything that cannot be derived directly by other attributes. For example, you cannot give me the term based on the course code. That requires using natural joins.

3.3 Union, Intersection, and Difference in SQL

These can be expressed as:

- `(subquery)UNION (subquery);`
- `(subquery)INTERSECT (subquery);`
- `(subquery)EXCEPT (subquery);`

If you are using queries in queries, you wrap them in parentheses and you don't put a semicolon at the end of a subquery. **The result from the three queries above will always be a set!! By default.**

But there are bag versions of union and intersection.

When writing views inside the parentheses, you MUST actually type `select * from <VIEW>`

3.3.1 Bag operations

Bag operations put the `all` keyword at the end. They are not on by default.

- `Union all` keeps all: if we treated bags like python lists, we get `bag1 + bag2`
  - add type operation
- `Intersect all`: keep the number of copies common to both. If set A has 4 instances of x and set B has 3 instances of x, the intersection will have 3 instances
  - `min()` type operation
- `Except all`: the elements on the right side are willing to explode and self-destruct if they see something equal to the one the left side, but one may only take one
  - max of subtract type operation, 0 type behavior.

3.4 Views

Names holding queries. Useful for compacting stuff.
`CREATE VIEW <view name> AS <query>`. Views persist until I drop them.

We can rename columns:
`CREATE VIEW <view name>(col1name, col2name, ...) as ...`

Use `drop view <view name>` to get rid of it. There are two types of views:

- **Virtual**
  - Attaches a name to the query (we'll only use this)
  - Every time you reuse this value, you will have to query again
  - This means that if the table updates, you always get an up-to-date table
- **Materialized**
  - Constructs and stores the results of the query. Expensive to maintain! (PostgreSQL didn't support this for a long time)
  - We won't use this
  - It's great if the underlying tables don't change very much

To use a view as a query, you must wrap it around with `(select * from <view>)` otherwise it will not work.

3.4.1 View use cases

- Break down a large query
- Provided another way of looking at the same data for one category of user

3.5 Joining in SQL

The result of a join could make keys no longer keys, because your result is a different table, which should mean something different in context.

We have these types of joins:

- `CROSS JOIN` (same as the comma in SQL)
- `NATURAL JOIN`
- `JOIN ON`
  - Theta join (join with given condition)
- `LEFT JOIN (ON CONDITION)`
- `RIGHT JOIN (ON CONDITION)`
- `FULL JOIN (ON CONDITION)`

Left join ensures that we keep all the rows on the left and fill the missing correspondences on the right with NULL. If we see all the rows on our resulting table from the left tuple, then we know that we have performed a `LEFT JOIN`.

We use theta joins (`JOIN ON`) instead of `NATURAL JOIN` as schemas tend to change **to help us catch errors when making queries when schemas do change**.

3.6 Subquery as a value in Where

$$\sigma_{\text{value} > \langle \text{a query} \rangle}$$

You cannot put a subquery in a condition in relational algebra, but you can in sub-queries like normal in SQL, which only works fine if your sub-query returns a 1×1 table. You will get an error otherwise if your subquery is larger than that (no results count as `NULL`), as you cannot run comparisons with multiple results.

But if you want to do a comparison with **multiple** values, we can require that:

- Value we are comparing is greater than every value, or
- Greater than at least one value (ANY/SOME)

3.6.1 Any/Some $\exists$

The syntax for ANY/SOME (these two words are interchangeable) is:

- `X <comparison> ANY (<subquery>)`

This evaluates to true if and only if the comparison holds for at least one tuple in the subquery result:

$$\exists y \in \langle \text{subquery results} \rangle \text{ such that } x \ \langle \text{comparison} \rangle \ y$$

3.6.2 All $\forall$

- `X <comparison> ALL (<subquery>)`

This evaluates to true iff the comparison holds for every tuple in the subquery result.

We could rewrite this query using `max` instead.

3.6.3 In $\in$

Syntax:

- `x in (<subquery>)`

True if x is in the set of rows given by the subquery

3.6.4 Exists

Syntax:

- `Exists (<subquery>)`

True if and only if the subquery has at least one tuple (is not empty).

Even if the tuple is null.

3.7 Making your own tables

Want to create something like

Level	Average
Easy	69
Hard	32

Use union and selecting a name:

```
(SELECT "Easy" as level, avg(...) from [...])
UNION
(SELECT "Hard" as level, avg(...) from [...])
```

Remember that `SELECT` happens last. `SELECT "Easy" as level` creates a column named `level` and just populates all the rows as `"Easy"`.

3.8 Search Paths

The top level is called `public`. `SET search_path to <SCHEMA>` is like changing the directory. Otherwise, use dots (`.`) to delimit directories and tables. The default schema is `public`, and you cannot nest schemas inside schemas. It is a flat structure.

```
1 table university.student
```

I can show search paths by using `show search_path`

If I want to remove schemas:

`Drop schema <SCHEMA> cascade;`

(Use `cascade` to make sure everything inside of it is dropped so no errors occur. To prevent error message, you must make sure it exists so use `if exists`). That's why on the top of a lot of files, you would see:

- Drop the schema if it exists, and cascade down
- Re-create it
- Set the search path to the schema

3.9 Workflow

- Create a DDL file with the schema
- Create a file with inserts to put content in the database (duplicate keys are NOT allowed! Under any circumstances, even in the bag concept.)
  - Tuples do not need keys, but if they do then you outright cannot have duplicates.
- Import these in PostgreSQL
- Run queries directly in the shell or by importing queries written in files

4 Pyscopg

SQL is not Turing complete. There's a trade-off:

- It's optimized for certain things
- But it can't do everything

You cannot control the format. End users (not necessarily with programming knowledge) should not be writing queries.

If we combine SQL with a language like Python, we can solve these problems, but SQL is based on relations and conventional relations don't have that. We'll use `psycopg2`. This allows us to connect from inside the Python program out to databases (like PostgreSQL).

When using `py s copg2.connect`:

- Your database name is `csc343h-UTORID`
- Your user is your `UTORID`
- Don't put anything into the password field

4.1 Creating Queries

First, open a cursor. It allows the Python program to pass information back and forth to the SQL database. Then, you can pass in SQL queries directly from Python strings. We **do NOT recommend this.**

```
1 # ... some opening code first: pyscopg2.connect(...)
2 cur = conn.cursor()
3 cur.execute("SELECT name, netWorth FROM MovieExec;")
4 for row in cur:
5     name = row(0)
6     worth = row(1)
7     # do something interesting with name and worth on the
          python side
8 conn.close()
```

If this code changes anything, it won't be reflected after close as nothing is committed. You will have to `conn.commit()`

Sometimes, you may not know the queries ahead of time. Then:

- Hard code the parts of the queries you know
- Use string substitution

The `cur.execute` has a second argument, like format strings in C. This method sanitizes the strings to prevent mischief.

DO NOT USE F-STRINGS OR ANY BUILT-IN STRING OPERATIONS (+) WITH USER-INPUTTED STRINGS INSIDE ARGUMENTS FOR `cur.execute`, AS THIS COULD CAUSE SECURITY PROBLEMS

For example, a malicious user could input `John' ; delete from please_do_not_delete; --` and the `'` is something you do NOT want. `P ys copg` automatically sanitizes these types of mischief.

```
1 cur.execute("%s", (to_place_here,))
```

4.2 Merge Conflicts

In Pyscopg, nothing persists until you commit. This is different than the SQL shell, who commits right after a semicolon (by the way, your session is preserved when you quit PSQL unless you reload your database).