CSC343 Notes

Introduction to Databases

https://github.com/icprplshelp/

Last updated April 18, 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, $\sigma_{\text{GPA}>3.7 \wedge \text{HS}<1000,\ \text{major}=CS}$
- `project` ($\pi_{\text{attribute}_1,\text{attribute}_2,\ldots}$ (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

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

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)
    1. 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`

Be careful when using `*` in conjunction with `GROUP BY`, as due to how `*` evaluates, unless you are querying from a table with one column, you will get an ambiguous attribute error.

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 Set operations (NO DUPLICATES)

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>`

3.4 Bag operations (ALL, DUPLICATES)

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.

3.5 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.

3.5.1 View use cases

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

3.6 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**.

`NATURAL LEFT JOIN` is valid syntax.

3.7 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)

THESE COMPARISONS ARE ROW-WISE. WE ARE COMPARING ONE ROW TO MANY OTHERS. This means a 2-column operation would feel like `(2, 3)in [(2, 3), (4, 5), (6, 7)]`.

3.7.1 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:

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

3.7.2 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.

3.7.3 In $\in$

Syntax:

- `x in (<subquery>)`

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

3.7.4 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.

3.8 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"`.

3.9 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.

```
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 messages, 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

3.10 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

3.11 Defining Types in Table Attributes

When creating a table in SQL, you **must** define the type of that variable. Here are some types:

- `CHAR(n)`
  - Fixed length string of n characters, padded with blanks
- `VARCHAR(n)`
  - Variable length string up to n characters.
- `TEXT`
  - Free-for-all string
- `INT` or `INTEGER`
- `FLOAT` or `REAL`
- `BOOLEAN`
- `DATE`
- `TIME`
- `TIMESTAMP`

SQL defines a string enclosed in 'single quotes', and NOT smart quotes.

3.12 Defining our own type (Constrained Types)

Defined in terms of a built-in type. You can constrain them or give them a default value. For example:

```
CREATE DOMAIN Grade as int
Default null
Check (value >=0 and value <= 100)
CREATE DOMAIN State as varchar(4)
Default 'ohio'
Check (value in ('ohio', 'oklahoma', 'wyoming'))
```

3.12.1 Default Values

When no value has been specified.

3.13 Primary Keys

Every table can either have 0 or 1 primary keys (but all tables should have one). It is the key to the table. You can never have more than one value of it (it is unique), and it **cannot** be null. If you declare something to be a primary key, its value can't be null.

That is, the `PRIMARY KEY` – a set of one or more attributes for a relation. You can declare primary keys:

```
Create table TBL
ID integer primary key,
...;
```

Or

```
Create tabke TBL (
...,
primary key(this, that));  -- means (this, that), a pair,
   is a key, like dept, cNum
```

A TABLE CANNOT HAVE TWO PRIMARY KEYS. Do not use the primary key keyword twice. If you want a key that is made of two parts, then you do have to declare it in the last line. `Primary key(dept, cNum)` (same syntax for `unique` as well).

3.14 Uniqueness Constraints

When an attribute is `UNIQUE` for a relation, it means:

- They can form a key (unique and no subset is)
- Multiple attributes can be declared unique
- Nulls are never equal to each other (primary keys don't allow null values anyway)

```
1 CREATE table tbl(...,
2 Unique(ID, ID2));  -- for multiple attributes in a key
```

For example, you can use `Unique(firstname, lastname)`. This will prevent two insertions of the same `firstname`-`lastname` pair.

In the same table, an attribute can be labelled as a primary key, and another attribute can be labelled as unique. If I have a default value for a unique, if I do two blank insertions, the program generates an error.

Note that `unique` per attribute is DIFFERENT than `unique(a1, a2)`. The second one `unique(a1, a2)` restricts pairs. The first one just restricts duplicates if I project only that property.

3.15 Foreign Key Constraints

```
1 Foreign key (sID) references Student
```

Means that `sID` in this table is a foreign key that references the primary key of the table Student.

- Every value for `sID` in this table must actually occur in the `Student` table (keys in this table $\subseteq$ those in the student table).

- `sID` must be a primary or unique key there

3.15.1 The References Keyword

```
1 Create table people (
2 SIN integer primary key,
3 Name text,
4 OHIP text unique);
5 Create table Volunteers (
6 Email text primary key,
7 OHIP text references People(OHIP));
```

Here, for each row in Volunteers, `Volunteers[text]` $\subseteq$ `people[text]`

3.15.2 Referring table example

Took refers to both offering and student. It has `sID`s and `oID`s. If I remove a student from the student table, then the references within the took table about that student no longer makes sense. So we should remove them or do something.

I either put in a table that doesn't refer to anything in the home table, or a deletion in the home table that makes something in the referring table make no sense.

3.15.3 Safe and Risky Operations Regarding Foreign Key Constraints

So what can I do, and what can I not do?

- Safe to delete from `Took`
- Risky to insert and update anything in/to `Took` (what if the `sID` is not in `Student`?
- Safe to insert into `Student`
- Risky to delete or update anything inside in/to `Student`

3.16 Reaction Policies - Enforcing Foreign Key Constraints

How could a database react to a risky operation?

1. Prevent it from happening
2. Allow the operation if it doesn't break constraint rules
3. Propagate the change to the affected tables such that no constraint rules are broken
    4. Deleting or updating from `Student` will also delete/update all matching `sID`s from `Took`.
4. Propagate the change, but instead of deleting, set any instance of the deleted key to `NULL`
    5. If affected columns aren't primary keys, then we can set to `NULL`

Because of asymmetry, any affected table decides what would happen (the owner/designer of the table).

When does the DBMS bring the rules here? When we try to insert a tuple. If we try to insert a tuple that violates the foreign key constraint, then there is a violation. We respond it by specifying a reaction policy. These help us deal with insertions in the referring table that violate key constraint or changes in the home table.

If you don't say anything in the reaction policy, then the program gives you an error when you try to violate a key constraint.

If you say something, then something should be done in the referring table to get rid of the violation.

Primary	Unique
One of a kind	One of a kind
No nulls allowed	Can have nulls; nulls are never equal to each other

3.17 Types of Reaction Policies

A reaction policy can be set up on a table to either block unsafe queries or update/change at most **itself**. It keeps its hands off all other tables.

- `cascade`: propagate the change to the referring table
- `set null`: set the referring attributes to null
- default: forbid the change

Suppose `R` refers to `S`.

- Doing anything unsafe to `R` (insert) cannot modify `S`, so it will prevent the query from being made
- Doing anything unsafe in `S` (delete or update) **can** modify `R`.

This means if anything is referred to, change it can change the things that refer to it.

3.17.1 What I can react to

All scenarios in **the table that is looking at another table:**

```
on delete -- each row: if the table I'm looking at deletes this, I will
on update -- each row: if the table I'm looking at updates, I will
on delete restrict on update cascade
```

Why no insert? Because inserting to a table that is referred to has no effect on foreign relationship. And regardless, I am not allowed to modify the table I'm looking at, so violating a constraint by inserting into the table that refers to another is not allowed.

Changing constraints:

```
ALTER TABLE took
DROP CONSTRAINT took_sid_fkey;
ADD CONSTRAINT took_sid_fkey
    FOREIGN KEY (sid)
    REFERENCES student(sid)
    ON DELETE CASCADE ON UPDATE CASCADE;
```

Setting a value to null counts as an update.

3.17.2 An instance of trying to update

```
1 update student set sid = 111 where sid = 11111;
2 ERROR: update or delete on table "student" violates the
     foreign key constraint...
```

3.18 Save points and Rolling Back

When I write `begin;`, I create a transaction. If I make a mistake, I have to roll back to the safe point. I can also roll back fit I do something I didn't mean to.

I type `ROLLBACK;` to undo my changes.

- `begin;` to start a transaction or a safe point
- `rollback;` if an error occurs or if you want to undo and exit the transaction
- `commit;` once you're done and want to save and exit the transaction

If I'm NOT in a transaction, every query (after a `;`) commits.

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 `pyscopg2.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 import pyscopg2
2 conn = psycopg2.connect("dbname=csc343h-UTORID user=UTORID
     password=")  # could also use py kwargs
3 cur = conn.cursor()
4 cur.execute("SELECT name, netWorth FROM MovieExec;")
5 for row in cur:
6     name = row[0]
7     worth = row[1]
8     # do something interesting with name and worth on the
        python side
9 conn.close()
```

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

A cursor can hold the results of a query and allow us to iterate over them.

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. `Pyscopg` 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).

5 Privileges and Transactions

- Roles can be assigned to one or more users to determine what permissions they have.
- The user that initiates a session owns it. Whether they initiate it through sql or by running a python script.
- Schemas and their contents are owned by the suer who creates them.
- SQL checks whether the session owner has sufficient privileges to carry out a task.

Owners can create roles and assign (groups of) users to them. Superusers (Postgres) and owners can grant privileges to roles, and revoke them.

For each query or role, you want to grant the minimal privilege necessary for a role to do its job. You don't want to show someone everyone's home addresses, right?

5.1 Adjusting Privileges

Owners will always be able to `DROP`, `GRANT` `REVOKE`, and so on. These are special privileges.

To grant a privilege **TO**, use:

```
GRANT <OPERATION> ON <some_object> TO <user>;
GRANT <OPERATION> (specific_column) ON <some_object> TO <
   user>;
```

To revoke a privilege **FROM**:

```
REVOKE <OPERATION> ON <some_object> FROM <user>;
REVOKE <OPERATION> (specific_column) ON <some_object> FROM
   <user>;
```

Do not mix up to and from.

Objects can be schemas and tables.

Ordinary privileges are as follows:

- `SELECT` (`r` "read")
  - Allows `COPY` to
- `INSERT` (`a` "append")
  - Allows `INSERT` and `COPY FROM`
- `UPDATE` (`w` "write")
  - Allows `UPDATE`
- `DELETE` (`d`)
  - Allows a row to be deleted. Oftentimes requires `SELECT` as well.

- `TRUNCATE`
  - Allows this operator
- `REFERENCES`
  - Allows foreign key constraints to be created on this table
- `TRIGGER`
  - Allows creation of this on a table
- `CREATE`
  - For databases, allow new schemas and publications to be created.
  - For schemas, allow new objects to be created within. To rename an object, you must own it and have this privilege for the schema.
- `CONNECT`
  - Allows connection
- `TEMPORARY`
  - Allows creation of temporary tables in this database
- `EXECUTE`
  - TBA
- `USAGE`
  - TBA
- `ALL`
  - All of them

Most privileges require the `SELECT` privilege to work completely as intended. Creating a view does not run the query, so SQL will not stop me from creating a view. You can revoke privileges on views.

If you don't have any privileges, if you try to look for a schema, you wouldn't be able to see it.

5.2 Accessing Privileges

`\dp` (display privilege) displays privileges. Access privileges display:

```
1 name=vector of access control characters/who gave those
    permissions`
```

Ex: `user=arwdDxt/givenby`

5.3 Granting Some Privileges

```
1 grant select(dept, cnum) on course to name;
```

Gives me select permissions for just `dept` and `cnum`. If I try to select anything else, or my queries involve attributes that I don't have the permissions for, I wouldn't have permission for it. And I certainly can't look at `*`. Rule of thumb: if you can CTRL+F the words, then you should probably have been given permissions for them.

```
grant select(dept, cnum, instructor)on offering to name;
grant select(dept, cnum, breadth)on course to name
```

5.4 Constraints and Foreign keys

`update` maintains constraints, and `select` can help verify constraints.

5.4.1 Inserting or updating a table that references another table

When inserting, we need to be able to verify if the foreign key constraint is met, so we must be able to see specific attributes in the tables that the table we're inserting to is referencing.

There are no issues when deleting.

5.4.2 Deleting or updating a referenced key

I must be able to perform all reaction policies, and I must be able to know that there's a reaction policy in the first place. Databases have NO loopholes.

Note that I do not need permission to know that I don't have permission to do something.

5.4.3 Deleting in general

REQUIRED AT ALL TIMES

When deleting, you need to grant `DELETE` on the entire table, because you are really deleting by rows. This means that when you delete a row from a table that other tables are referencing, you need to have deleted or update perms on all the tables that reference the table I'm deleting from.

6 Transactions

6.1 Concurrent users

Single statements in a PSQL shell begins with a snapshot of the DB the MOMENT we hit ENTER and either succeeds or fails (and rollbacks) after the statement finishes executing.

However, we want transactions (groups of statement) to either all succeed or all fail, for correct behavior (in other words, atomicity).

6.2 Problems with Transactions

6.2.1 Dirty read

Using uncommitted changes from other sessions; not allowed in PSQL

6.2.2 Non-repeatable reads

View of DB items can change between the start and end of transaction due to other transactions **committing** (I get balance, someone else changes balance, then I subtract from the balance)

6.2.3 Serialization anomaly

Occurs when the result of executing transactions T_1 and T_2 concurrently is **not** (T_1 then T_2) or (T_2 then T_1)

6.3 Levels of Isolation

The SQL standard has 4 levels of isolation. Postgres only has the strictest three, so this avoids reading uncommitted.

```
1 SET TRANSACTION ISOLATION LEVEL serializable
```

6.3.1 Read committed (default)

- Only committed results from other transactions are seen in the transactions
- When a row is being modified by a transaction T_1, it locks those rows the moment I submit a query that modifies those roles. Other transactions T_2 that try to UPDATE (not read) an uncommitted, but modified row by T_1 will lock (wait) until it gets committed by transaction T_1.
- When you try to delete a row, and a block occurs, the rows that are deleted are rows who meet the delete condition before and after I run the delete command

6.3.2 Repeatable read

During a transaction, we don't see anything that was committed from start to end except for the transaction itself

6.3.3 Serializable (strictest)

If your results are different than your transactions happening in either order (permutations of transactions running in order), then committing will fail

The other one risks dirty reads

6.4 Costs vs. Certainty

Isolation levels come at a cost. The `serializable` level has substantial overhead, and it is very costly for the DBMS. Mostly, use the default.

7 Functional Dependencies

Source

Functional dependencies are a generalization of the notion of keys. They allow the system to store data more easily. They can be used to reason about query and do query optimization. This allows declarative queries to be executed efficiently.

Suppose we have two relations:

```
Student(SSN, sName, address, HScode,
        HSname, HScity, GPA, prio)

Apply(SSN, cName, state, date, major)
```

Suppose that `priority` is determined by GPA:

- GPA $> 3.8 \rightarrow$ priority $= 1$
- $3.3 <$ GPA $\leq \rightarrow$ priority $= 2$
- GPA $\leq 3.3 \rightarrow$ priority $= 3$

If this relationship is guaranteed, then any two tuples that have the same GPA has the same priority. Formalizing this:

$$\forall t_1, t_2 \in \text{student} : (t_1.\text{GPA} = t_2\text{GPA}) \Rightarrow (t_1.\text{prio} = t_2\text{prio})$$

This logical statement is the definition of a functional dependency and we would write this as

$$\text{GPA} \rightarrow \text{Priority}$$

7.1 Generalizing Functional Dependencies

$$\forall t_1, t_2 \in R : (t_1.A = t_2A) \Rightarrow (t_1.B = t_2B)$$

Means in R, $A \rightarrow B$.

Functional dependencies don't always need to have one attribute on each side. They can have to.

If $A_1, A_2, \ldots$ and $B_1, B_2, \ldots$ are attributes of relation R, we can instead use:

$$\forall t_1, t_2 \in R : (t_1[A_1, \ldots, A_n] = t_2[A_1, \ldots, A_n]) \Rightarrow (t_1[A_1, \ldots, A_n] = t_2[A_1, \ldots, A_n])$$

$$A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_n$$

Means that if two tuples t_1 and t_2 has the same values for all of $A_1, A_2, \ldots, A_n$, then they must also have the same values for all of $B_1, B_2, \ldots, B_n$.

I may abbreviate $\bar{A} = A_1, A_2, \ldots$ and the same for $\bar{B}$.

7.2 Creating Functional Dependencies

They are based on knowledge of the real world. All instances of relation must adhere. Meaning if we have:

$$\bar{A} \to \bar{B} \quad R(\bar{A}, \bar{B}, \bar{C})$$

This means R would look like this:

$\bar{A}$	$\bar{B}$	$\bar{C}$
$\bar{a}$	$\bar{b}$	???
$\bar{a}$	$\bar{b}$	?!?

If the a values are the same then so does the b values.

For `student`:

$$\text{SSN} \to \text{sName}$$

This means we require the identifier to uniquely identify the student name. And also:

$$\text{SSN} \to \text{address}$$

This means that social security number can't link to more than two addresses.

$$\text{HScode} \to \text{HSname}, \text{HScity}$$

The same high school code can't be associated with an ambiguous high school name or city.

Functional dependencies can be in the other direction:

$$\text{HSname}, \text{HScity} \to \text{HScode}$$

And if we assume we have one GPA for each student, then we have

$$\text{SSN} \rightarrow \text{GPA}$$

$$\text{GPA} \rightarrow \text{Priority}$$

Functional dependencies are **transitive**, so this automatically implies

$$\text{SSN} \rightarrow \text{Priority}$$

Functional dependencies really depend on real-world data. For example, **if all college names are unique,** then we can state that every application to a college has a single date. If that were the case, then cName $\rightarrow$ date.

If we want students to only apply to one major, then SSN, cName $\rightarrow$ major.

7.3 Functional Dependencies and Keys

7.3.1 "Is a key" condition

If $\bar{A} \rightarrow$ all other attributes, then $\bar{A}$ is a **key.**

7.3.2 Trivial FDs

We have a trivial functional dependency if this statement holds:

$$\bar{A} \rightarrow \bar{B} \text{ AND } \bar{B} \subseteq \bar{A}$$

7.3.3 Nontrivial FDs

We have a nontrivial functional dependency if this statement holds:

$$\bar{A} \to \bar{B} \text{ AND } \bar{B} \not\subseteq \bar{A}$$

7.3.4 Completely nontrivial FD

We have a completely nontrivial FD if this holds:

$$\bar{A} \to \bar{B} \text{ AND } \bar{A} \cap \bar{B}$$

7.3.5 In summary

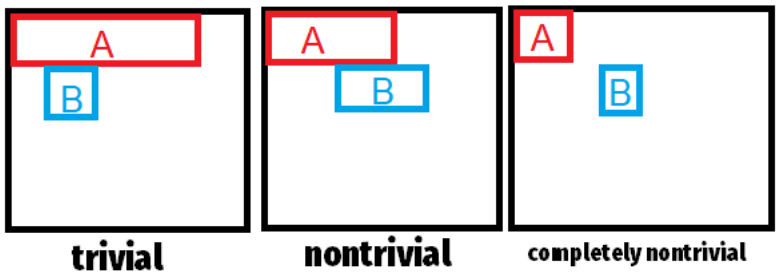

Figure 1: Functional dependency diagram

7.4 Rules for FDs

7.4.1 Splitting/combining Rule

Feels like the distributive property (splitting is $\Rightarrow$, combining is $\Leftarrow$)

$$(\bar{A} \to B_1, B_2, \ldots, B_m) \iff (\bar{A} \to B_1 \text{ AND } \bar{A} \to B_2 \text{ AND } \ldots)$$

This does NOT work the other way around:

$$(A_1, A_2, \ldots, A_n \to \bar{B}) \not\Rightarrow (A_1 \to \bar{B} \text{ AND } A_2 \to \bar{B} \text{ AND } \ldots)$$

(The combining rule still works for this, so the $\Leftarrow$ works)

Why doesn't this work? Because if `HSname` and `HScity` $\rightarrow$ `HScode`, then we don't want to have `HScity` standalone, as this removes a lot of information when it comes to choosing the HS.

7.4.2 Trivial Dependency Rules

If we have a trivial dependency (the following hold):

$$\bar{A} \rightarrow \bar{B} \text{ AND } \bar{B} \subseteq \bar{A}$$

THEN both the following hold:

$$(\bar{A} \rightarrow \bar{B}) \Rightarrow (\bar{A} \rightarrow \bar{A} \cup \bar{B})$$

$$(\bar{A} \rightarrow \bar{B}) \Rightarrow (\bar{A} \rightarrow \bar{A} \cap \bar{B})$$

7.4.3 Transitive rule

The $\rightarrow$ operator is transitive.

$$(\bar{A} \rightarrow \bar{B} \text{ AND } \bar{B} \rightarrow \bar{C}) \Rightarrow (\bar{A} \rightarrow \bar{C})$$

This is really obvious if you try to look at it, or draw an example.

7.5 Closure of Attributes

(Everyone follows itself)

Given relation, a set of FDs, and a set of attributes $\bar{A}$. I want to find all B such that $\bar{A} \rightarrow B$.

The "all B" is known as the closure. Notationally, the closure of $\bar{A}$ is $\bar{A}^+$, or $\{A_1, A_2, \ldots, A_n\}^+$

> This means that $\bar{A}^+$ is all attributes, from our functional dependencies, that can be unambiguously determined by $\bar{A}$

How do I find that? Here's the algorithm:

- Start with $A_1, \ldots, A_n$
- Add attributes to that set until I get to that closure. REPEAT UNTIL NO CHANGE:
  - If $\bar{A} \rightarrow \bar{B}$ and $\bar{A}$ in the set above, then add $\bar{B}$ flattened to the set

7.5.1 Running the closure algorithm

If we are given:

```
Student(SSN, sName, address,
        HScode, HSname, HScity,
        GPA, priority)

SSN -> sName, address, GPA
GPA -> priority
HScode -> HSname, HScity
```

And we want to find $\{\text{SSN}, \text{HScode}\}^+$

Just add attributes that are functionally determined in this set. This algorithm feels like a graph search algorithm. This is how I would do it:

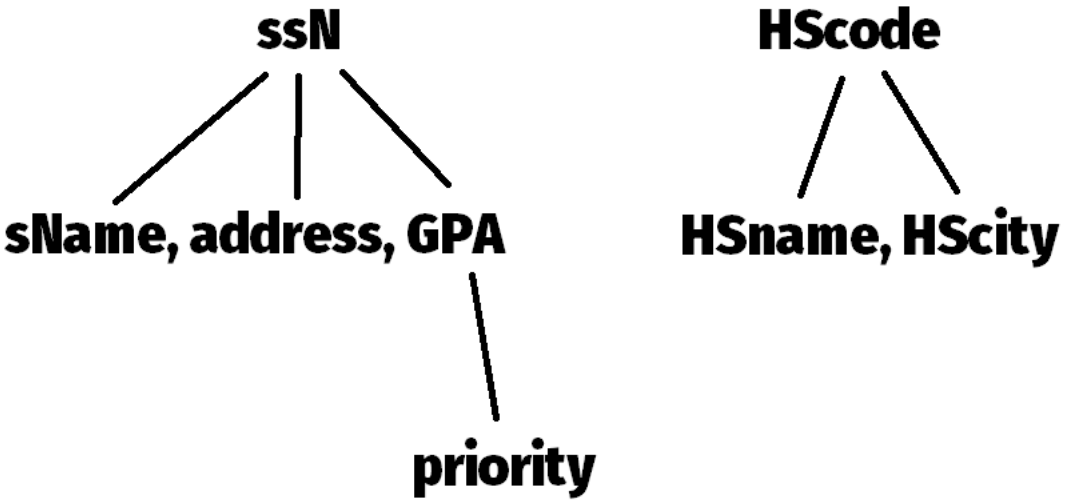

Figure 2: Flatten this tree and all words we see are elements in the closure

Apparently, from this, `SSN` and `sName` are able to unambiguously determine all other attributes of a student (if recorded). This means that `{SSN, sName}` form a **KEY.**

7.5.2 Closure to determine Keys: verification

Is $\bar{A}$ a key for R? We can use closure to do that.

1. Compute $\bar{A}^+$
2. If $\bar{A}^+ =$ all attributes, then $\bar{A}$ is a key.

7.5.3 Closure to determine Keys: finding all keys

Brute force the algorithm above with every possible subset of attributes in our relation. Note that this doesn't give me the primary key alone; it gives me all superkeys as well.

To run this more efficiently, consider running this algorithm in increasing size of attribute subsets. This means the real algorithm would rather

- Test if each single attribute is a key, then
- Test if each pair of attributes are keys
- And so on

(Stop when we find one?)

7.6 Following from

- S_1 and S_2 are sets of FDs.
- S_2 **follows from** S_1 if every relation instance satisfying S_1 also satisfies S_2.

For instance:

$$S_2 : \{\text{SSN} \to \text{priority}\}$$
$$S_1 : \{\text{SSN} \to \text{GPA}, \text{GPA} \to \text{priority}\}$$

How do we test this?

Does $A \to B$ follow from S? We can use closure:

1. $\bar{A}^+$ based on S, check if $\bar{B}$ is in the set.
2. Use Armstrong's Axioms.

7.7 Specifying FDs for a relation

We want: **minimal** set of **completely nontrivial FDs** such that all FDs that hold on the relation **follow from** the **dependencies in the set.**

Get this by actually making FDs. You'll need practice.

7.8 Projecting FDs

We have a big table. We break them up to small tables. What are the FDs from our original table that apply in our new tables?

We'll have to project our functional dependencies onto the attributes of the new table.

We start with

- `S`, a set of FDs
- `L`, a set of attributes. $L \subseteq S$
- We want to return the projection of `S` onto `L`:
  - All FDs that follow from `S` and involve only attributes from `L`.

```
def Project(S, L):
    Initialize T to { }
    for each subset X of L:
        compute X+ # the closure of X
        for every attribute A in X+:  # subset
            If A is in L:  # alt: filter a not in L after
                add X->A to T  # union
    return T
```

7.8.1 Speed-ups for this algorithms

- No need to add $X \rightarrow A$ if A is in X itself. It's trivial and is a waste of time. These subsets of X won't yield anything, so no need to compute their closures:
  - The empty set
  - The set of all attributes

The big save:

If we find that $X^+ =$ all attributes (or a super key), then we don't have to look at anything that is a superset of that superkey, as any other supersets will be a super key. That can save a great deal of time.

Projection is expensive. Suppose R_1 has n attributes. There are 2^n subsets of R_1 . We have to cover ALL of them.

7.9 Minimum Basis

if we have a set of FDs (or equivalent FDs)

A minimum basis is an equivalent set of FDs that:

- has no redundant FDs
- no FDs with unnecessary attributes on the LHS

```
def MINIMAL_BASIS(S):
    split the RHS of each FD
    for each FD X -> Y where |X| >= 2, can you remove an
       attribute from X and get an FD that follows from S,
       do so (it is stronger), replacing its previous
        # stronger because the original FD follows from the
            new one
    for each FD f:
        If S - {f} => f:
            Remove f from S.
```

This is a case that I will **like** to split on the left. Whenever I split on the left, the split FDs may not all be equivalent, but they **are stronger.**

Keep in mind:

$(A \to B) \Rightarrow (XAB \to B)$ where X is one or many attributes.

So when you remove X from XAB, you DO have to check if AB follows from S first because we don't know yet.

8 Functional Dependencies, but concise

Information on functional dependencies that as short as possible

8.1 Closure

$\overline{A}$'s closure is everything that can be inferred from $\overline{A}$.

To find the closure of $\overline{A}$, assume you know $\overline{A}$ and perform the domino effect.

8.2 Follows From

To see if LHS $\rightarrow$ RHS follows from S, a set of existing FDs, find the closure of LHS, and see if it contains all of RHS.

8.3 Projection

To project a set S of FDs onto a set of attributes $\overline{X}$:

For each subset of $\overline{X}$, find the closure of it, and infer functional dependencies from it. For instance, add $X \rightarrow X^+$ to our ever-accumulated new projection set. Get rid of any redundancies, or better skip them.

Additional tips about efficiency:

- For maximum efficiency, start with the singletons and work your way up. If your singleton A ends up inferring everything, it is a key and you do not need to look at any subsets of $\overline{X}$ that contain A.
- If your singleton A ends up inferring everything but B, you do not need to look at any subsets of $\overline{X}$ that contain $A \cup B$.
- Note that I didn't need to use the word "singleton" for my last point. A and B could be multiple attributes each.

8.4 Minimal Basis

To find **a** minimal basis of a set of FDs S:

1. Split the RHS of each FD in S (makes it easier to work with)
2. Try to reduce the LHS for each FD in S
    1. For each subset of the LHS, if it can be inferred by everything else, remove it from the LHS. Start from checking what the singletons can infer, and work your way up.

3. Figure out if we can remove entire FDs
    1. For each FD: can I infer its RHS from anything but the FD? If so, get rid of it. Once you get rid of it, it's gone for good, so do not use it from that point onwards.

9 Database design

Try not to stuff all your information into one table. For example:

A table that stores:

- A part
- Manufacturer
- Manufacturer address
- Seller
- Seller Address
- Price

It is a waste of space (redundant) to store attributes you tend to see repeated many times in a table, in a table with a lot of columns.

Another thing is that manufacturer address will always be consistent with manufacturer address, making the fact that manufacturer address there be redundant.

Key lesson: **AVOID REDUNDANCY.** Redundant data can lead to anomalies.

- If a manufacturer moves and we only update one tuple, the data is inconsistent. So it's not just the space, it's the work we have to Dom and **the potential sources of error**. **That's why we have helper functions.**
- part $\rightarrow$ manufacturer, so do we really need the manufacturer data anyway?
- Every seller has one address: seller $\rightarrow$ seller address

That's why we use functional dependencies. We can only know what FDs apply through context of the domain.

We call FDs functional because there is a mathematical function that takes a value for X and gives a unique value for Y. It is just a lookup table – not a function.

The rule for the mathematical function is unknown, it come from domain knowledge. Maybe we could call the function `dk(LHS)-> RHS`.

When we write a set of CDs, we mean that all of them hold. But we can write sets of CDs in equivalent ways. When we say that S_1 is equivalent to S_2, we mean that S_1 holds $\Leftrightarrow$ S_2 holds.

Note that

$(A \rightarrow B \text{ AND } AB \rightarrow C) \Rightarrow (A \rightarrow C)$

9.1 Functional Dependencies to Redesign tables

To improve a badly design schema, just decompose them such the two A, B:

- attribute unions up to the original
- joins recreate the original

So, the attributes of your two tables should not be disjoint. Decompositing it into BCNF (Boyce-Codd normal form) guarantees it. This ensures no redundancies, and guarantee that a new schema does not exhibit any anomalies.

9.2 BCNF

Relation $R \in$ BCNF: if for every nontrivial FD $X \rightarrow Y$ that holds in R, X is a super key. Otherwise, BCNF is violated.

> In other words, all LHSes are super keys. This means that the closure of X must contain everything, for each FDs X $\rightarrow$ Y.

BCNF requires that only things that functionally determine everything can functionally determine anything.

Nontrivial means that Y is not contained in X. A super key does not need to be minimal.

There is an algorithm for BCNF decomposition:

```
BCNF_decomp(R: Relation, F: Set[FD]) -> None:
    if an FD X->Y in F violates BCNF:
        Compute X+
        Replace R by two relations with schemas:
            R1 = X+  # if R1 is BCNF, R1 is our map
            R2 = R - (X+ - X)
        Project the FDs F onto R1 and R2 # project 1, check
            project another, check... first violator,
           decompose on that
        Recursively decompose R1 and R2 into BCNF
```

In RA (no duplicates), any relation with no more than two attributes is in BCNF.

Concisely:

- Given relation R and a set of FDs S:
- Go through each FD in S, inspect its LHS X, and check if it's a superkey. If not, then split R into two:
  - X^+
  - Not X^+, but include X itself so natural join can reconstruct the two
- Project the FDs onto your two splits. Ideally, project, check, project, check for BCNF. If fail, then recursively decompose again.

9.3 Speed-ups

- Don't need to know any keys, super keys only matter.
- When projecting FDs onto new FDs, does the new relation violate BCNF because of this FD? If so, stop projecting (short circuit), as you will discard this relation and decompose further.
  - And the FD that violates it is the one you want to split on (because you already know it does and I don't want to waste computational power).

9.4 Redundant information

If I could have just inferred another attribute by looking at an attribute, if I had the map for all FDs.

9.5 WHAT WE WANT: Prosperities from decomposition

1. No anomalies
    2. No redundancies. Non-BCNF risks redundancies.
2. Lossless join
    3. It should be possible to project the original relations on the decomposed schemas
    4. Then reconstruct the original by joining. We should get back exactly the original tuples
3. Dependency preservation
    4. All the original FDs should be met regardless of how I mess around with my decomposed table
    5. Keep all the rules about the domain.

We'll see that if we follow the algorithm for BCNF decomposition, you should meet [2] from above. But you may not always be able to get all the FDs back with BCNF decomposition ([3] is not met).

This means that BCNF decomposition may cause data loss from the original FD. **This means if I were to recklessly put in data into by BCNF split tables and join them together, it could violate an FD that was in place before I did the BCNF split.**

The BCNF property alone does not guarantee lossless join. You need to start with your WHOLE relation and follow your decomposition algorithm to guarantee lossless join. We can check for this using the chase test.

- So WHOLE relation in terms of the entire schema? Is this context dependent?

9.6 Lossy join

For any (not just BCNF) decomposition, it is the case that

$$r \subseteq r_1 \bowtie \ldots \bowtie r_n$$

But this may not always be true

$$r_1 \bowtie \ldots \bowtie r_n \subseteq r$$

In other words, $r_1 \bowtie \ldots \bowtie r_n$ can give us spurious tuples. Note that BCNF decompositions still satisfy the second condition.

9.7 3NF (Third Normal Form)

3NF is a more relaxed version of the BCNF condition.

X $\rightarrow$ A violates 3NF $\iff$ X is not a superkey and A is not prime (in a key – the minimal superkey).

In other words, it's okay if X is not a super key as long as A is prime

A key is a minimal superkey. If you know something is a superkey, but none of its strict subsets can be a key/superkey as well, then your superkey IS a key. You may need context for this… perhaps ask more about this?).

3NF guarantees that when you join everything you won't get garbage.

3NF does not guarantee no redundancies (because its not BCNF, and if anything violates BCNF it is possible for redundancies to occur).

There is a tradeoff:

- We either have redundancies
- We lose FDs

The choice depends on context. No one knows how to get all three. It has not been shown to be impossible.

9.7.1 3NF Synthesis

Given set of attributes and FDs:

1. Make sure our FDs are a minimal basis
2. Every FD becomes its own relation
3. Make sure SOME relation is a superkey
4. If not, add a relation whose schema is some key

9.7.2 Synthesis vs. Decomposition

3NF is a bottom up generation where we have our attributes and FDs, but we do not start off with any relation.

For BCNF, we start with a schema and break it down.

9.8 Finding all keys

Objective: find ALL the keys. Exhaust them.

A key is a super key that can't be reduced further.

If we have relation `ABCDEFG` and FDs $\{A \to BE, C \to BFG, G \to AC\}$

Then we have the following schemas:

- ABE, CBFG, GAC

What could be a key?

- €?
  - To test, find C^+, and see if it functionally determines everything? $C^+ = CBFGAE$. This is missing D, so it cannot functionally determine everything.
- CD?

  - $CD^+ = CBFGAED$, which tells us that CD is a super key. Is this a key? We know that C is not a super key, but we need to check the closure of D: $D^+ = D$, so CD is minimal. So CD functionally determines everything, and no subset of it does.

- GD?
  - $GD^+ = CDACBEF$. This is also a super key. D is not a super key, but $G^+ = GACBEF$ (missing D). We have another key. We've found two keys.

9.8.1 Checking if we actually found all possible keys

It's not a good approach to intuitively try to find all keys. We can't make any algorithmic guarantees. How do we know that we have all keys?

1. Exhaust every combination (that requires $2^n - 1$ checks)
2. The following:

For each attribute, looking at all FDs:

On LHS at least once	On RHS at least once	Conclusion
Yes	No	Must be in every key. Could give me more informatic
No	Yes	Won't be in any key
Yes	Yes	Need to check
No	No	Must be in every key

9.8.2 Example

For instance, for attributes `ABCDEFG` and FDs $\{A \to BE, C \to BFG, G \to AC\}$

On LHS at least once	On RHS at least once	Conclusion
Yes	No	
No	Yes	BEF $\Rightarrow$ in no key, ignore
Yes	Yes	ACG $\Rightarrow$?
No	No	D $\Rightarrow$ in every key

This means that we'll have to find the closure of all subsets of ACG, starting from the smallest subsets first as if you find a singleton that is a key, you can ignore all supersets of that as they're not minimal.

(ACG)

For these type of tables, if you find that an attribute (union ...) is a super key, then you do not need to check all supersets of that attribute.

Attribute $-$ everything in every key	Closure	Conclusion
A	$AD^+ = ADBE$	Not a superkey
C	$CD^+ = ABCDEFG$	Superkey (C isn't a key and so isn't D, so key also)
AG	omitted	superkey, and also a key
AC	skip this as AD is a superkey	–
AG	skip this	–
CG	skip this	–
ACG	skip this	–

This means we only checked 3 closures out of 127.

Note: we didn't check the empty subset of everything on both LHS + RHS, but if we had any attributes on the LHS only, then we should check the set of all attributes on no RHS alone as a candidate.

If $H \rightarrow C$ was a FD that held:

- H must be part of every key
- and we would check if HD on its own is a key
  - if not, keep checking
  - if so, you can stop

9.9 Chase Test

How do we check if a decomposition has:

1. No redundancy?
2. Preserves FDs?
3. Has lossless join?

10 ER Diagrams

10.1 Removing Redundancies From ERs

- A thing deserves an entity set
- A detail about some other thing deserves an Attribute

An entity set should satisfy at least one of the following conditions:

- It is more than the name of something and has at least one non-key attribute
- Or it is the "many" in a 1-N or N-N relationship
  - If you have many values for an attribute, how are you going to represent that? If one person owns multiple cars, you should use an entity set. If a person can only own one car, that may not be needed.

So those cases you wouldn't use an attribute. Keep the key as the attribute, and the name of it if the username can't be unique.

10.2 Null values

Minimize things that invite null values.

10.3 Weak Entity Sets

There is **no global authority** capable of creating unique IDs

It is unlikely that there could be an agreement to assign unique student numbers across all students in the world

So we need at least information about student number and the institution they're at.

However, **it is usually better to create unique IDs**. Such as SIN, automobile VIN, and so on. Before I can submit taxes, I need to show that my SIN corresponds correctly.

Anything that to uniquely identify it depends on another relationship / entity.

10.4 Selecting a Primary Key for Entities

Relations borrow keys from 2+ entities, but we can tweak this a bit, as if one of the entities is participating with cardinality 1-1, then that forms a key for the relationship, so it simplifies a relationship key. But so far, relationship keys are the keys collected for every entitity that uses them.

For entities:

- Can't risk NULL values
- Prefer if the attribute is single
- Internal keys preferable to external ones
    - weak entities depend for their existence on other entities

- A key that is used by many operations to access instances of an entity is preferable to others

10.4.1 Avoid multi-attribute and string keys

Reason 1: They are wasteful

- Movies with (title, year) as the key
- Can we use `movieID`s? It saves 75% space and a lot of typing

Reason 2: They break encapsulation

- Prevents information leaks: if we have the ID of a patient and not the name, we don't need to leak the patient name
- Can be done with permissions
- Basic principle of privacy legislation (who needs to see what?)

Reason 3: they are brittle

- Duplicates, lack of identification, and changes

Computers are also really good at integers

10.5 Attributes with cardinality > 1

The relational model does not allow multi-valued attributes, so they must be converted to entity sets. That is the standard way to break that up:

```
Company -(1, N)- Relationship (Posesses)-(1, 1)- Phone
```

If a company has multiple phone numbers, then you'll have to create a relationship. Note the `(1, N)` on the Company cardinality. That `many` cardinality on the Phone becomes the many cardinality (a.k.a. need for list) of the company participating in the relation `Possesses`.

11 ER Models to Logical Schema

Starting from an E/R schema, an equivalent relational schema is constructed. Equivalent means a schema capable of representing the same information. A good translation should not allow redundancy and not invite unnecessary null values. Sometimes, NULLs are needed but we don't want more than we need.

11.0.1 First Approximation

Make all entities into relationships

Make each relationship into a relation or a table. Take the keys from the entities that connect to that relationship.

Now how do we simplify that? If you do it according to above your result will be messy.