Outline

1. More Logistics
2. Introduction to Databases
3. Introduction to the Relational Model
4. Relational Algebra

Piazza

We will use Piazza for class discussion. If you are enrolled or waitlisted, you should have received an email with the link.

If you are waiting for a PTE, please enroll in the Piazza:

https://piazza.com/ucla/spring2018/comsci143/home
Waitlist Update

I am actively working with the Department on dealing with the waitlist. Please sit tight.

If you are on the waitlist, a PTE is not necessary.

Ryan R. Rosario
rrosario@cs.ucla.edu
Computer Science 143: Introduction to Database Systems
UCLA Department of Computer Science

PTE Update

I’ve decided I’d like to wrap up this whole process sooner rather than later because of CCLE restrictions. Once we get the waitlist figured out, I will start distributing PTEs.

On the sign in sheet, if you’ve already provided your information, just check off the PTE box. I don’t need your information again. Thanks!

Ryan R. Rosario
rrosario@cs.ucla.edu
Computer Science 143: Introduction to Database Systems
UCLA Department of Computer Science

Discussion/Recitation/Lab Section Update

The TAs are figuring out how to staff the discussion sections and we will let you know what they decide.

Go to your recitation section this Friday and your TA will be there.

Ryan R. Rosario
rrosario@cs.ucla.edu
Computer Science 143: Introduction to Database Systems
UCLA Department of Computer Science

Project 2

After discussing with the TAs, we have decided Project 2 will involve Spark likely involving writing an ETL job. More info to follow.
Last Time...

Last time we discussed the problems with using a flat file system to store data. We also talked about how UCLA student records used to use a flat file system and some of the problems it faced when transitioning to DB2 (corrupted files).

- **Data Integrity and Redundancy**: differing preferences formats, languages among sysadmins lead to possible data inconsistencies, and storage format natively and needlessly duplicates data (though this saved the day for UCLA).
- **Sysadmin is the Bottleneck**: all system changes, course changes, restrictions must be encoded in a custom application.
- **Lack of Atomicity**: system failures in the middle of a multi-step process can result in lost data. (i.e. bank transfer, exchanging classes). we should do it perfectly, or not at all.

So we concluded there must be a better way, and that better way is...

- **Concurrent Access**: It's possible for data to get out of sync when multiple people read/write concurrently (i.e. two student enrollment activities reading the same enrollment cap, incrementing it, and overwriting the old value concurrently, yielding the wrong count.)
- **Security**: Without proper filesystem controls, flat files can be read by anybody with access to the filesystem.
There Must be a Better Way!

A Database

For most of this course, this means an RDBMS, for Relational Database Management System.

Database Purpose

A database abstracts away how the data is stored, maintained and processed. We usually don’t care how the data is laid out on disk.

- Provides a way to view, add, update and delete data without worrying about files and breaking data integrity.
- Provides ONE single location for all data in the database (though the database itself may be distributed over multiple nodes, we don’t care).

Levels of Abstraction

- **Physical.** How the data are stored (if we care). If you’ve worked with MySQL before, this is encapsulated in the two engines MyISAM and InnoDB.
- **Logical.** Describes what data there are and the relationships among the data.
- **View.** What the user sees. It is typically just a part of the database, such as used to generate a report. In the flat file case, the user relied on the sysadmin to “see” the data.

A database (more precisely a table or schema, which we will discuss in a bit) is a way of abstracting a structured type, like you’d see in C++.

---

\[\text{struct Section} \{
\text{int srs; // unique ID number for this section}
\text{int cap; // max number of students allowed}
\text{int instructor_uid; // TA or instructor info}
\text{int parent_srs; // link to the lecture}
\}\]
Instances and Schemas

The information stored in a database at a particular point in time is called an *instance*. This terminology has been resurrected by its use with Amazon Web Services.

More importantly, the overall *design* of a database is called a *schema*.

A *subschema* refers to the design of a particular part of the database, but we usually still call it a schema, if it refers to a DB or a table.

This is analogous to variable declarations and classes containing variables. Each realized class object is called an *instance* of the class. Each variable has a set data type, similar to a database schema.

Instances and Schemas (contd.)

Data Model

The data model contains "conceptual" tools for describing data, their relationships, semantics and consistency constraints.

There are five major types.

Data Model (contd.)

- **Relational.** Uses a collection of *tables* to represent data and their relationships. Each table has columns with unique names and a data type. Each column represents an attribute, and each row represents a record.

<table>
<thead>
<tr>
<th>bldg</th>
<th>floors</th>
<th>nearby_squirrels</th>
<th>resident_rats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boelter</td>
<td>9</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Engineering VI</td>
<td>5</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>SAC</td>
<td>3</td>
<td>1024</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Analogous to a matrix (sort of), and an R dataframe or Excel spreadsheet, roughly speaking.
Data Model (contd.)

- **Entity-Relationship (ER)**. Uses a collection of basic objects called entities and relationships among them. We typically use this to visualize a database design.

![Entity-Relationship Diagram](image)

Object-Oriented. Draws a strong analogy to object-oriented programming with encapsulation, methods and object identity.

- A more rare beast. CS 241A covers these databases in detail.
- Data are essentially treated as instances of classes rather than tables.
- Examples include Java Data Objects (JDO), Gemstone, Caché, Versant.

Data Model (contd.)

- **Semi-Structured**. Individual data objects may have different sets of attributes.
  - Very different from the relational model, but many common semi-structured data types can be converted into the relational model.
  - JSON and XML are two examples of data types that are natural for the semi-structured model.
  - Very important to know how to use these data formats. If you take CS 144 you will see this again.
  - We will also discuss examples. Most area called NoSQL databases.

- **JSON**

```
{
  "query": {
    "count": 1,
    "created": "2018-04-01T19:46:28Z",
    "lang": "en-US",
    "result": {
      "channel": {
        "condition": {
          "code": "28",
          "temp": "54°F",
          "time": "2018-04-01T19:46:28Z";
        }
      }
    }
  }
}``

"Etwor"
Data Model (contd.)

XML

```xml
<result>
  <channel>
    <item>
      <title>Weather Condition</title>
      <updated>2018-04-25T08:00Z</updated>
      <link>http://xml.weather.yahoo.com/xmlrss/1.0</link>
      <description>
        [source: @title for @link] [source: @capability for @link]
      </description>
    </item>
  </channel>
</result>
```

Network/Hierarchical/Graph. What is old is now new again.

- Actually pre-dates the relational model.
- Defines data instances as nodes, and relationships between nodes as edges.
- Was considered unwieldy as they tied data very closely to database implementation details. This is still the case.
- One notable NoSQL example is neo4j.

"Database Languages"

There are two main semantic systems for working with databases:

- **Data Definition Language (DDL)**
- **Data Manipulation Language (DML)**

For the relational model, they are usually both SQL.
“Database Languages”

It is pronounced as either *Es kyew ell* or *See-kwell* or *Sequel* but I won’t judge.

Data Manipulation Language (DML)

Databases use one of two types of

- **Procedural**: User specifies what data are needed and *how* to get it.
- **Declarative**: User specifies what data, but not *how* to get it.

This follows the same semantics as with theory of programming languages (CS 131). SQL is **declarative**. SQL takes some features from Datalog (a subset of Prolog, another declarative language).

Datalog is still in niche use. The *Cascading*\(^a\) data processing framework actually implements Datalog in Clojure\(^b\) as *Cascalog*\(^c\).

\(^a\) https://www.cascading.org/
\(^b\) https://clojure.org/
\(^c\) http://cascalog.org/
Data Manipulation Language (DML)

A **query** is a written expression to retrieve or manipulate data.

A **query language** is simply the language it is written in.

Note that JavaScript can be a query language!

---

**SQL in Particular**

A SQL query takes one or a pair of tables and outputs a single table.

We can also perform operations on only one table. Let's take the fictional building fauna table from a few minutes ago. Suppose UCLA wants to hire an exterminator and send them to particular buildings. We can execute a query on just this one table.

```
1 SELECT bldg
2 FROM campus_fauna
3 WHERE resident_rats > 0;
```

We can also use multiple constraints. Perhaps we need to give the exterminator a list of tall buildings, buildings with more than 2 floors as the method he/she uses to solve the problem is different:

```
1 SELECT bldg
2 FROM campus_fauna
3 WHERE resident_rats > 0 AND floors > 2;
```

---

SQL is not a procedural language and is not as powerful as the general Turing machine.

There are computations that cannot be done in SQL, for example, sequential or iterative computations typically used in mathematical fields. Or algorithms that require the user to specify how to perform a computation.

And even if it could be done, it would be unwieldy. **So what can we do?**
When SQL isn’t Enough

The most common option:

Write an ETL job in your favorite language to extract the data from the database using a database connection driver such as ODBC for C (Open DB Connectivity) or JDBC (Java DB Connectivity), transform the data using your favorite language and then use the driver to write the data (load) the data into a new table. Pretty much every language has drivers for the most common databases.

If you don’t need the resulting data written back to a database, you can eliminate the load step. Another method is to dump the data to disk in a convenient format and load it into your favorite language, but this is manual and accident prone.

Data Definition Language (DDL)

There are several types of constraints a DDL can specify:

- **Domain Constraints** restrict the values of a particular column to a particular type, or a particular set of values. Checked on any queries that manipulate data.

- **Referential Integrity**. There are many situations where a value in one table must appear in some other table (or multiple other tables) for the data to be considered complete.
Data Definition Language (DDL) (contd.)

- **Assertions.** Both 1 and 2 are types of assertions, but there are several other constraints we can impose. In the student records case, we can impose some example constraints that are checked by the DBMS on each attempted data manipulation:
  - “A student may not take more than 21 units in a given quarter.”
  - “A student may not repeat a class unless their grade was C- or lower during the previous attempt.”
  - and many more.

Authorization. We can also restrict access to databases and schemas as well as particular operations on these databases and schemas. We can apply these rules to users, groups, etc. The most common database privileges are:
  - Inserting into tables.
  - Reading or selecting data from a table.
  - Updating the values in a table.
  - Altering the structure of a table.
  - Deleting data.
  - Creating and deleting tables and databases.

Data Storage and Querying

Databases also have a **storage manager** that allows us to typically not worry about how the data is laid out on disk.

Databases can be on the order of GBs to TBs. Some organizations have even exceed the PB milestone.

Most of the data cannot fit in RAM, and must be read in from disk. This must be done efficiently because reading from the disk is very slow.

- **Authorization and data integrity checks** to prevent unauthorized access and enforce integrity constraints.
- **Transaction management** ensures the database remains in a consistent state despite a system failure, and that concurrent accesses are handled appropriately. A transaction is a series of operations that must be executed as a logical unit, in a particular order, for success.
- **File manager** allocates and manages disk space.
- **Buffer manager** deals with swapping data from disk to RAM and back.
Data Storage and Querying

The storage manager uses a few of its own data structures

- **Data files** which store the... well... data.
- **Data dictionary** containing metadata about the structure and schema of the database.
- **Indices** that enable optimized and efficient lookup of data. (i.e. looking up an instructor name from a UID)

One data structure you have used in programming that can be used as an index in your code is the **hash table** (called a **hash map** in Java or a **dict** in Python).

**What is another one?**

**Interview question: How would you store/search 1 billion phone numbers?**

**ANSWER:**

It depends on what we want to do with the phone numbers! If all we want to do is lookup a particular phone number, either for existence, or to get its “owner”, a hash table is ideal as it gives $O(1)$ retrieval. But what if we are trying to autocomplete a phone number? We could perhaps use a **self-balancing binary search tree (BST)** or even better, a **trie**. Perhaps a trie yields better performance because a BST needs to consistently assess and assign new root values (typically the median).

Point being, you have worked with data structures that work like indices, and some database indices are based on variations or specializations of these.

There is also a **query manager**.

When a query is executed the DML (i.e. SQL) statements are organized into a **query plan** that consists of low-level instructions that the query evaluation engine understands to perform some operation on the data.

The query plan is very important for debugging problems in queries, particular queries that run slowly or use a lot of resources. We will discuss how to use the query plan with the **EXPLAIN** syntax later.
We can now put together all of these pieces into the bigger picture. From your text:

We strive to avoid duplicated data in a database. We will talk more about this later in the quarter, but it’s something good to keep in mind. For now, suppose we have the following three tables, as is:

<table>
<thead>
<tr>
<th>CourseOfferings</th>
<th>Instructor</th>
<th>Email</th>
<th>Affiliate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>Course</td>
<td>Title</td>
<td>Instructor</td>
</tr>
<tr>
<td>199</td>
<td>CS 118</td>
<td>Networks</td>
<td>Ryan R. Rosario</td>
</tr>
<tr>
<td>199</td>
<td>CS 111</td>
<td>Operating Systems</td>
<td>La, S.</td>
</tr>
<tr>
<td>165</td>
<td>CS 143</td>
<td>Database Systems</td>
<td>Kampes, M.</td>
</tr>
</tbody>
</table>

We can replace redundant data with simple IDs (integers) and use join together the tables to create the CourseOfferings table.

Course IDs to course codes and titles is a simple mapping that can be encapsulated in a fact table called MasterCourseCatalog. We can do the same with instructors: instructor ID mapping to a fact table about instructors (name etc.).

We can reduce our tables to the following...
Brief Comment on Database Design: Duplicated Data

This process of deduplicating data is called *normalization* and we will talk about it later.

If this is all confusing, don’t worry, we will discuss it. These comments are just to motivate where we are headed.

Database Architectures

There are two main architectures for database systems:

- Server/client
- Distributed

Server-Client Architecture

All data is stored in one centralized database, and clients access this database directly.

Distributed Architecture

In a distributed architecture, there are several database servers. They may all contain identical data (replicated) or different parts of the data (sharded) that depend on geography or some other circumstance.

Each server may coordinate with a master database, or may exchange data amongst themselves (eventual consistency) or via caches (real-time access use case).
Distributed Systems: Shedding Tiers

The centralized case can be realized as containing two tiers: the client and the server interact via some client application containing a database access library call.

It can also be realized as containing three tiers. The client calls a client application, which makes a network call to a network application server. The network application server in turn communicates with the database system itself. The most common examples of a three-tier architecture involves a web server like Apache or Nginx, or perhaps some kind server exposing a REST API.

Some business intelligence tools, for example, can serve as either the application client or the application server.
What is a Table

A relational database consists of a collection of tables.

Each table consists of rows and columns:

Each row is a record, with a set of attributes that are related to that record.

Each column represents a particular attribute, has a unique name, and a particular data type.

An Example: YouTube Videos and Comments

Below appears a few rows from a table containing information on a large number of US YouTube videos that were crawled by a researcher. The researcher did the same thing for YouTube videos from Great Britain.

<table>
<thead>
<tr>
<th>video_id</th>
<th>title</th>
<th>channel</th>
<th>cat</th>
<th>views</th>
<th>likes</th>
<th>dislikes</th>
<th>cm</th>
</tr>
</thead>
<tbody>
<tr>
<td>XIPV6Z1Gjo</td>
<td>1 YEAR OF VLOGGING</td>
<td>Logan Paul Vlogs</td>
<td>24</td>
<td>4094629</td>
<td>320053</td>
<td>5931</td>
<td></td>
</tr>
<tr>
<td>cLdxuaxaQwc</td>
<td>My Response</td>
<td>PewDiePie</td>
<td>22</td>
<td>7645890</td>
<td>575697</td>
<td>39774</td>
<td></td>
</tr>
<tr>
<td>Ayb2qbZHm4</td>
<td>Honest College Tour</td>
<td>CollegeHumor</td>
<td>23</td>
<td>859289</td>
<td>34485</td>
<td>726</td>
<td></td>
</tr>
<tr>
<td>EVp4-qXYoJE</td>
<td>Chargers vs. Broncos</td>
<td>NFL</td>
<td>17</td>
<td>743947</td>
<td>6126</td>
<td>352</td>
<td></td>
</tr>
</tbody>
</table>

Each row in the first table refers to an individual video. Each row in the second table refers to an individual comment on some video.

Videos are identified by a unique identifier, the `video_id` and comments are identified by a unique identifier, the `comment_id`.

Note that a row in the comment table is related to a row in the video table via the `video_id`.

And all columns/attributes in the same row are related to each other because each row represents an individual entity: a video or a comment.
Getting Relational

In the relational model theory, we call a table a relation.

We will call a column an attribute. We will call a row a tuple of \( n \) values (\( n \)-tuple) where \( n \) is the number of attributes.

A tuple is a data structure common in Python. Other languages have similar constructs by different names. It is just an ordered set of values.

And the youtube_comment relation contains the following attributes

- video_id
- comment_id
- comment
- likes
- replies

So, the youtube_video relation contains the following attributes

- video_id
- title
- channel
- cat_id
- views
- likes
- dislikes

Bottom Line: A relation is a set of tuples, where each tuple has the same number of attributes.

Yes, set as in set theory.
Getting Relational

Each attribute has a domain, a set of legal values. This domain can be discrete or continuous.

An attribute can have a null value meaning it is unknown or does not exist.

Relationship Status: It’s Not Too Complicated

So we have a set of tuples related to videos, and a set of tuples related to comments. But how do we uniquely distinguish among videos and among comments?

In other words, which attribute, or set of attributes, uniquely identify videos and comments?

Getting Relational

We can now begin creating a schema for our YouTube relations. It looks like this, but it will get more interesting as we go along...

```
youtube_video(video_id, title, channel, cat_id, views, likes, dislikes)
```

```
youtube_comment(video_id, comment_id, comment, likes, replies)
```

Note that video_id appears as an attribute in both relations. This is how we define a relationship between the tuples in the youtube_video relation and the youtube_comment relation.

Relationship Status: It’s Not Too Complicated

Video tuples are uniquely identified by video_id. There cannot be any duplicate video_ids in the relation.

Comment tuples are uniquely identified by comment_id. There cannot be any duplicate comment_ids in the relation.
A superkey is a set of one or more attributes that unique identifies a tuple and distinguishes it from all other tuples.

Superkey: A Fly in the Ointment

Exercise: Suppose for a moment that instead of having a unique comment_id for each comment, we assign sequential ID number starting at 1 for each comment, and we do this for each video.

<table>
<thead>
<tr>
<th>video_id</th>
<th>comment_id</th>
<th>comment</th>
<th>likes</th>
<th>replies</th>
</tr>
</thead>
<tbody>
<tr>
<td>cLhuxaQuc</td>
<td>1</td>
<td>Love you Pewdiepie don’t apologize your fine</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cLhuxaQuc</td>
<td>2</td>
<td>The N word is not okay to say because...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ayb2obDBen4</td>
<td>1</td>
<td>im watching this at college</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ayb2obDBen4</td>
<td>2</td>
<td>Yo this was funny af</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

What would the superkey be?

Answer: Both video_id and comment_id uniquely identify a tuple, they form a composite superkey.

<table>
<thead>
<tr>
<th>video_id</th>
<th>comment_id</th>
<th>comment</th>
<th>likes</th>
<th>replies</th>
</tr>
</thead>
<tbody>
<tr>
<td>cLhuxaQuc</td>
<td>1</td>
<td>Love you Pewdiepie don’t apologize your fine</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cLhuxaQuc</td>
<td>2</td>
<td>The N word is not okay to say because...</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ayb2obDBen4</td>
<td>1</td>
<td>im watching this at college</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Ayb2obDBen4</td>
<td>2</td>
<td>Yo this was funny af</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Suppose we use the sequential comment_id version of the youtube_comment relation where the superkey is video_id and comment_id. I can also include other attributes and it is a valid superkey, but it contains redundant information because video_id and comment_id are the only attributes we need to uniquely identify a tuple.

Suppose we drop video_id from K. comment_id is a proper subset of K, but it is not a superkey because each video with at least one comment will have a duplicated comment_id.

Thus, via proof by handwaving, comment_id and video_id are a composite superkey.

Primary Key

Social Security Number may be the obvious answer, but not everybody has a Social Security Number!

We can use SSN for people that have one, but for people who do not, we may need to use some other identifier. Perhaps we concatenate a country ID with a passport ID, or convert a retina/iris scan into some kind of hash key.

Point being: We must choose primary keys with care, and we must use attributes that never change, or change very rarely.
Primary Key

What about an index of every human on Earth?

Last time we talked about data privacy. Scientists are working diligently on decoding the human genome.

Could this be used as a primary key? Let’s hope not!

Foreign Key

For a particular relation $r$, a foreign key is an attribute that is the primary key of some other relation $r'$. Loosely speaking, a foreign key is an attribute that ties the tuples of two relations together into one. A foreign key in $r$ does not uniquely identify a tuple in $r$, but does in the referred relation $r'$.

Foreign Key

For the original implementation of the youtube_comment relation (call it $r$), video_id is the only foreign key. video_id is the primary key of the youtube_video relation $r'$.

In the sequential ID version of youtube_comment, video_id and comment_id form a composite primary key and video_id alone is a foreign key.

Foreign Key

Primary and foreign keys can also be the same in the trivial case when there is a one-to-one relationship between the relations.
Foreign Keys and Referential Integrity

Foreign keys are often used to satisfy referential integrity constraints.

Suppose we have two relations: residents and dorms, where each tuple in dorms contains a building identifier and a student identifier to denote the relationship “student resident $s$ lives in dorm $d$” and each tuple of residents contains various information about students that live in the dorms, such as roommate preferences.

The dorms relation:

<table>
<thead>
<tr>
<th>building_id</th>
<th>uid</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEN</td>
<td>111111111</td>
</tr>
<tr>
<td>DEN</td>
<td>123456789</td>
</tr>
<tr>
<td>CYN</td>
<td>012012012</td>
</tr>
<tr>
<td>HDK</td>
<td>246802468</td>
</tr>
</tbody>
</table>

The primary key of residents is uid and the primary key for dorms is a composite of uid and some building_id because those attributes uniquely identify each tuple. uid by itself is a foreign key of dorms. Thus, during the school year when the dorms are active (assume there are no empty ones) we expect a particular resident uid to be present in both the residents relation and the dorms relation.

It does not make sense for a dorm resident to not have an assigned dorm building or for a dorm building to not have any residents! Both of those situations violate referential integrity.
Representing a Schema Visually: Word Soup

Relations, attributes and keys (both primary and foreign) are represented visually in a diagram.

- A box for each relation with the name of the relation at the top of the box, with the background of the name sometimes shaded and separated from the rest of the box with a horizontal line;
- Attributes are listed beneath the name of each relation, primary keys listed first, and underlined, followed by all other attributes.
- An arrow is drawn from each foreign key to the primary key of the referenced relation.

Representing a Schema Visually: An Example

For the original YouTube example, we would have the following. We draw an arrow from video_id in comments to video_id in videos but note that foreign key DOES NOT need to have the same name as the primary key in the other relation. We could call it whatever we want as long as the relationship is encoded properly.

Our New Schema

We can now describe our schema for both relations as follows:
youtube_video(video_id, title, channel, cat_id, views, likes, dislikes)
youtube_comment(comment_id, video_id, comment, likes, replies)

We will learn a better way to represent schemas once we start learning SQL.
The relational algebra is a semantic system used for modeling operations on relational data developed by Edgar F. Codd of IBM. It provides a theoretical foundation for the relational model as well as query languages such as SQL.

Perhaps unsurprising, relational algebra was largely unknown outside of Pure Mathematics until 1970 when Codd published a paper on the relational model of data.\(^a\)

Time to Get Mathy!

We do not want to spoil all of the fun, so we will just introduce the basics and go into more detail next lecture.

If you have not already done so, it would be a good to skim chapter 6, but don’t get bogged down in the details, as it can be a pretty dense read. We will go into more detail next lecture.

Remember that relations are sets.

Relational Algebra

All relational databases provide a set of operations that can be performed on a single relation, or on a pair of relations. We will introduce most of them today, and see the rest next time.

We can work with multiple relations, by chaining operations on pairs of relations. This is how we will do it in SQL, unsurprisingly.

- Select \( \sigma \)
- Join (and Natural Join \( \bowtie \))
- Cartesian Product (\( \times \))
- Set Union (\( \cup \))
- Projection (\( \Pi \))

Select \( \sigma \)

Select retrieves a particular tuple from a single relation that satisfies a particular constraint (if provided by the user) and returns a new relation which is a subset of the original relation.

An example of select as a SQL implementation:

```
SELECT title
FROM youtube_videos
WHERE likes > dislikes
AND views > 1000000
AND cat_id = 24
```
Select $\sigma$

**Select** retrieves a particular tuple from a single relation that satisfies a particular constraint (if provided by the user) and returns a new relation which is a subset of the original relation.

A common inefficiency in the SELECT statement:

1. `SELECT *`  
2. `FROM youtube_videos`  
3. `WHERE likes > dislikes`  
4. `AND views > 1000000`  
5. `AND cat_id = 24`

Retrieves unneeded columns, takes longer. Unless you are just exploring your data, do not use `SELECT *`.

Join and Natural Join $\Join$

A **join** combine two relations by merging pairs of tuples, one from each relation, into a single combined tuple. A **natural join** is an implicit join where the RDBMS joins the relations using common attribute names.

Joins are very powerful, and many people find them confusing, so we will dive deep in this topic later. There are several types of joins.

**Cartesian Product $\times$**

The **cartesian product** combines tuples from two relations, **but all pairs of tuples from the two relations are returned regardless if attributes match**. This is very different from the natural join where names of attributes must match.

Recall that if we have two fictional sets $S_1 = \{a, b, c\}$ and $S_2 = \{d, e\}$, the cartesian product

$$S_1 \times S_2 = \{ad, ae, bd, be, cd, ce\}$$

Formally,

$$A \times B = \{(a, b) | a \in A, b \in B\}$$
Set Union $\cup$

Set union $\cup$ performs a set union on two relations *that have the same structure*. It is a concatenation of the two sets of tuples.

Projection $\Pi$

An odd operator is the *projection* operator. It sounds more complicated than it is. Loosely speaking, we can imagine it this way: we can project a series of $n$ attributes into $k$ dimensions ($k < n$) by simply discarding $n - k$ of the attributes. This is exactly what the projection operator does. Given a relation $R$ and a subset of attributes $a_1 \ldots a_n$:

$$\Pi_{a_1 \ldots a_n} (R) = \{ t[a_1, \ldots, a_n] : t \in R \}$$

Set Union $\cup$ (contd.)

In the YouTube data example, the researcher also collected data on videos from Great Britain, but we only used the US data. If instead we wanted to combine the tuples from both relations, we could do something like the following. Assume `youtube_videos_us` is the US video data and `youtube_videos_gb` is the Great Britain video data.

1. `SELECT * FROM youtube_videos_us UNION -- or UNION ALL
2. `SELECT * FROM youtube_videos_gb`

That wasn’t too bad. We will dig in more next time.
Be On The Lookout

Homework 1 will likely be distributed Friday or Monday. You will have a week to work on it.

Project 1 specifications will go live this weekend. **Find your partners!**