Lecture #01

Relational Model & Algebra
COURSE LOGISTICS

Course Policies + Schedule: Course Web Page
Discussion + Announcements: Piazza
Homeworks + Projects: Gradescope
Final Grades: Canvas

Do **not** post your solutions on Github.
Do **not** email instructors / TAs for help.
We do **not** control the waitlist.

Admins will move students off the waitlist as spots become available.

This class will be offered in Fall’24 too!
LECTURE RULES

Do interrupt for the following reasons:

→ I’m speaking too fast.

→ You don’t understand what I’m talking about.

I’ll will not answer questions about the lecture immediately after class.
C++ REQUIREMENT

All the projects are in C++ …. If you are new to C++, you must pick it up quickly… If you can take and get all the questions on the following quizzes right, you are all set:

Scoping: https://www.learncpp.com/cpp-tutorial/chapter-7-summary-and-quiz/
Type Conversion: https://www.learncpp.com/cpp-tutorial/chapter-10-summary-and-quiz/
Lvalues/rvalues: https://www.learncpp.com/cpp-tutorial/chapter-12-summary-and-quiz/

... take it upon yourself to catch up ...

... also https://db.in.tum.de/teaching/ss23/c++praktikum/slides/lecture-10.2.pdf?lang=en

C++ Bootcamp: This Friday 1/19 from 3:30pm-4:30pm in GHC 6115
PROJECT 0: GOALS

→ Get you started on C++, so you are not surprised later.
→ Get you thinking about algorithms and concurrency.
→ P0 is about Conflict-Free Replicated Data Type (CRDT) – a distributed data structure that coordinates changes and produces an “eventually” consistent state.
→ Will connect to P0 with the last few topics in the class that is all about transactions)
→ I wanted an excuse to introduce CRDTs. As members of the data tribe, we should all know about this concept.

That is you by the end of this semester.

If you can’t score 100% on P0, you can’t stay in this class.
TODAY’S AGENDA

Database Systems Background
Relational Model
Relational Algebra
Alternative Data Models
DATABASE

Organized collection of inter-related data that models some aspect of the real-world.

Databases are the core component of most computer applications.
Create a database that models a digital music store to keep track of artists and albums.

Things we need for our store:
→ Information about Artists
→ What Albums those Artists released
FLAT FILE STRAWMAN

Store our database as comma-separated value (CSV) files that we manage ourselves in our application code.

→ Use a separate file per entity.

→ The application must parse the files each time they want to read/update records.

<table>
<thead>
<tr>
<th>Artist(name, year, country)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wu-Tang Clan&quot;,1992,&quot;USA&quot;</td>
</tr>
<tr>
<td>&quot;Notorious BIG&quot;,1992,&quot;USA&quot;</td>
</tr>
<tr>
<td>&quot;GZA&quot;,1990,&quot;USA&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Album(name, artist, year)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Enter the Wu-Tang&quot;,&quot;Wu-Tang Clan&quot;,1993</td>
</tr>
<tr>
<td>&quot;St.Ides Mix Tape&quot;,&quot;Wu-Tang Clan&quot;,1994</td>
</tr>
<tr>
<td>&quot;Liquid Swords&quot;,&quot;GZA&quot;,1990</td>
</tr>
</tbody>
</table>
Example: Get the year that GZA went solo.

**Artist(name, year, country)**

"Wu-Tang Clan", 1992, "USA"

"Notorious BIG", 1992, "USA"

"GZA", 1990, "USA"

```python
for line in file.readlines():
    record = parse(line)
    if record[0] == "GZA":
        print(int(record[1]))
```
FLAT FILES: DATA INTEGRITY

How do we ensure that the artist is the same for each album entry?

What if somebody overwrites the album year with an invalid string?

What if there are multiple artists on an album?

What happens if we delete an artist that has albums?
FLAT FILES: IMPLEMENTATION

How do you find a particular record?

What if we now want to create a new application that uses the same database? What if that application is running on a different machine?

What if two threads try to write to the same file at the same time?
FLAT FILES: DURABILITY

What if the machine crashes while our program is updating a record?

What if we want to replicate the database on multiple machines for high availability?
A database management system (DBMS) is software that allows applications to store and analyze information in a database.

A general-purpose DBMS supports the definition, creation, querying, update, and administration of databases in accordance with some data model.
DATA MODELS

A data model is a collection of concepts for describing the data in a database.

A schema is a description of a particular collection of data, using a given data model.

Preview of the relational model

```
CREATE TABLE Artist(name VARCHAR, year DATE, country CHAR(60));
CREATE TABLE Album(Albumid INTEGER, name VARCHAR, year DATE);
```
DATA MODELS

Relational
Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value
DATA MODELS

Relational < Most DBMSs
Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value
DATA MODELS

- Relational
- Key/Value
- Graph
- Document / XML / Object
- Wide-Column / Column-family
- Array / Matrix / Vectors
- Hierarchical
- Network
- Multi-Value

← NoSQL
DATA MODELS

Relational
Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family
Array / Matrix / Vectors  ← Machine Learning
Hierarchical
Network
Multi-Value
DATA MODELS

Relational
Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value

← Obsolete / Legacy / Rare
DATA MODELS

Relational
Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value
DATA MODELS

Relational  ← This Course
Key/Value
Graph
Document / XML / Object
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value
Early database applications were difficult to build and maintain on available DBMSs in the 1960s.

→ Examples: **IDS**, **IMS**, **CODASYL**

→ Computers were expensive, humans were cheap.

Tight coupling between logical and physical layers.

Programmers had to (roughly) know what queries the application would execute before they could deploy the database.
Ted Codd was a mathematician at IBM Research in the late 1960s.

Codd saw IBM’s developers rewriting database programs every time the database’s schema or layout changed.

Devised the relational model in 1969.
A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

ABSTRACT: The large, integrated data banks of the future will contain many relations of various degrees in stored form. It will not be unusual for this set of stored relations to be redundant. Two types of redundancy are defined and discussed. One may be employed to improve accessibility of certain kinds of information which happen to be of great demand. When either type of redundancy exists, those responsible for control of the data bank should know about it and have some means of detecting any "logical" inconsistencies in the set of stored relations. Consistency checking might be helpful in tracking down unauthorized (and possibly fraudulent) changes in the data bank contents.

E. F. CODD
Research Division
San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

This paper is concerned with data systems which provide users with tree-structured files or slightly more general network models of the data. In Section 1, inequivalences of these models are discussed. A model based on a many relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user’s model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, languages of data, networks of data, relations, redundancy, redundancy, recovery, optimization, relational language, predicative calculus, security, data integrity

CR CATEGORIES: 574, 575, 577, 630, 632, 422, 439

1. Relational Model and Normal Form

1.1. Introduction

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Chey, a [4] and six papers of applications to data systems has been to the deductive question answering systems. Lavey and Morgen [5] provide numerous references to work in this area.

In contrast, the problems treated here are those of data independence—the independence of application programs and terminal activities from growth in data types and changes in representation—and certain kinds of data inconsistency which are expected to become troublesome even in noninteractive systems.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] prevailing in the non-inferential systems. It provides a means of describing data with its natural structure only—that is, without imposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusion, not the least of which is misunderstanding the derivation of constraints for the derivation of relations (see remarks in Section 2 on the "constraint trap"). Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

1.2. Data Dependencies in Present Systems

The provision of data description tables in mostly developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed without logically impairing some application programs is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of selections of data (as opposed to individual items). Those of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

1.2.1. Ordering Dependence. Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to be positioned in any desired order, others requiring each element to participate in several orderings. Let us consider these existing systems which either require or permit data elements to be stored in at least one total ordering which is closely associated with the hardware-determined ordering of address. For example, the records of a file containing parts might be stored in ascending order by part serial number. Such systems normally permit application programs to assume that the order of presentation of records from such a file is identical to (or a subset of) the
CODASYL

COBOL/CODASYL camp:
1. The relational model is too mathematical. No mere mortal programmer will be able to understand your newfangled languages.
2. Even if you can get programmers to learn your new languages, you won’t be able to build an efficient implementation of them.
3. On-line transaction processing applications want to do record-oriented operations.

Relational camp:
1. Nothing as complicated as the DBTG proposal can possibly be the right way to do data management.
2. Any set-oriented query is too hard to program using the DBTG data manipulation language.
3. The CODASYL model has no formal underpinning with which to define the semantics of the complex operations in the model.

The great debate: “The Differences and Similarities Between the Data Base Set and Relational Views of Data.”

ACM SIGFIDET Workshop on Data Description, Access, and Control in Ann Arbor, Michigan, held 1–3 May 1974
RELATIONAL MODEL

The relational model defines a database abstraction based on relations to avoid maintenance overhead.

Key tenets:

→ Store database in simple data structures (relations).
→ Physical storage left up to the DBMS implementation.
→ Access data through high-level language, DBMS figures out best execution strategy.
**RELATIONAL MODEL**

**Structure:** The definition of the database’s relations and their contents.

**Integrity:** Ensure the database’s contents satisfy constraints.

**Manipulation:** Programming interface for accessing and modifying a database's contents.
KEY CONCEPT: DATA INDEPENDENCE (DI)

Isolate the user/application from low level data representation.

→ The user only worries about application logic.

→ Database can optimize the layout (and re-optimize as the workload changes).

**Diagram:**

- **Application**
- **External Schema**
- **Logical Schema**
- **Physical Schema**
- **Disk**

Logical DI

Physical DI

Views (SQL)

Schema, constraints, … (SQL)

Pages, Files, B-trees, … (DB system internal)
A relation is an unordered set that contain the relationship of attributes that represent entities.

A tuple is a set of attribute values (also known as its domain) in the relation.

→ Values are (normally) atomic/scalar.
→ The special value **NULL** is a member of every domain (if allowed).

**Artist**(name, year, country)

<table>
<thead>
<tr>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>GZA</td>
<td>1990</td>
<td>USA</td>
</tr>
</tbody>
</table>

**n-ary Relation**

= Table with n columns
A relation’s primary key uniquely identifies a single tuple.

Some DBMSs automatically create an internal primary key if a table does not define one.

DBMS can auto-generation unique primary keys via an identity column:

**IDENTITY** (SQL Standard)

**SEQUENCE** (PostgreSQL / Oracle)

**AUTO_INCREMENT** (MySQL)
RELATIONAL MODEL: FOREIGN KEYS

A foreign key specifies that an attribute from one relation maps to a tuple in another relation.
**RELATIONAL MODEL: FOREIGN KEYS**

**Artist**($id$, name, year, country)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>102</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>103</td>
<td>GZA</td>
<td>1990</td>
<td>USA</td>
</tr>
</tbody>
</table>

**Album**($id$, name, artists, year)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>artists</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Enter the Wu-Tang</td>
<td>101</td>
<td>1993</td>
</tr>
<tr>
<td>22</td>
<td>St.Ides Mix Tape</td>
<td>???</td>
<td>1994</td>
</tr>
<tr>
<td>33</td>
<td>Liquid Swords</td>
<td>103</td>
<td>1995</td>
</tr>
</tbody>
</table>
# RELATIONAL MODEL: FOREIGN KEYS

**Artist**($id$, name, year, country)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>102</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>103</td>
<td>GZA</td>
<td>1990</td>
<td>USA</td>
</tr>
</tbody>
</table>

**Album**($id$, name, artists, year)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>artists</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Enter the Wu-Tang</td>
<td>101</td>
<td>1993</td>
</tr>
<tr>
<td>22</td>
<td>St.Ides Mix Tape</td>
<td>???</td>
<td>1994</td>
</tr>
<tr>
<td>33</td>
<td>Liquid Swords</td>
<td>103</td>
<td>1995</td>
</tr>
</tbody>
</table>
### Artist(id, name, year, country)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>102</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>103</td>
<td>GZA</td>
<td>1990</td>
<td>USA</td>
</tr>
</tbody>
</table>

### Album(id, name, artists, year)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>artists</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Enter the Wu-Tang</td>
<td>101</td>
<td>1993</td>
</tr>
<tr>
<td>22</td>
<td>St.Ides Mix Tape</td>
<td>???</td>
<td>1994</td>
</tr>
<tr>
<td>33</td>
<td>Liquid Swords</td>
<td>103</td>
<td>1995</td>
</tr>
</tbody>
</table>

### ArtistAlbum(artist_id, album_id)

<table>
<thead>
<tr>
<th>artist_id</th>
<th>album_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>101</td>
<td>22</td>
</tr>
<tr>
<td>103</td>
<td>22</td>
</tr>
<tr>
<td>102</td>
<td>22</td>
</tr>
</tbody>
</table>
**RELATIONAL MODEL: FOREIGN KEYS**

**Artist** (id, name, year, country)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>102</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>103</td>
<td>GZA</td>
<td>1990</td>
<td>USA</td>
</tr>
</tbody>
</table>

**ArtistAlbum** (artist_id, album_id)

<table>
<thead>
<tr>
<th>artist_id</th>
<th>album_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>101</td>
<td>22</td>
</tr>
<tr>
<td>103</td>
<td>22</td>
</tr>
<tr>
<td>102</td>
<td>22</td>
</tr>
</tbody>
</table>

**Album** (id, name, year)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Enter the Wu-Tang</td>
<td>1993</td>
</tr>
<tr>
<td>22</td>
<td>St.Ides Mix Tape</td>
<td>1994</td>
</tr>
<tr>
<td>33</td>
<td>Liquid Swords</td>
<td>1995</td>
</tr>
</tbody>
</table>
RELATIONAL MODEL: FOREIGN KEYS

Artist(id, name, year, country)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>102</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>103</td>
<td>GZA</td>
<td>1990</td>
<td>USA</td>
</tr>
</tbody>
</table>

ArtistAlbum(artist_id, album_id)

<table>
<thead>
<tr>
<th>artist_id</th>
<th>album_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>11</td>
</tr>
<tr>
<td>101</td>
<td>22</td>
</tr>
<tr>
<td>103</td>
<td>22</td>
</tr>
<tr>
<td>102</td>
<td>22</td>
</tr>
</tbody>
</table>

Album(id, name, year)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Enter the Wu-Tang</td>
<td>1993</td>
</tr>
<tr>
<td>22</td>
<td>St. Ides Mix Tape</td>
<td>1994</td>
</tr>
<tr>
<td>33</td>
<td>Liquid Swords</td>
<td>1995</td>
</tr>
</tbody>
</table>
User-defined conditions that must hold for any instance of the database. → Can validate data within a single tuple or across entire relation(s).

→ DBMS prevents modifications that violate any constraint.

Unique key and referential (fkey) constraints are the most common.

SQL:92 supports global asserts but these are rarely used (too slow).

**Artist(id, name, year, country)**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>102</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>103</td>
<td>GZA</td>
<td>1990</td>
<td>USA</td>
</tr>
</tbody>
</table>

CREATE TABLE Artist(
  name VARCHAR NOT NULL,
  year DATE,
  country CHAR(60),
  CHECK (date_trunc('year', year) = year));

CREATE ASSERTION myAssert
  CHECK ( <SQL> );
DATA MANIPULATION LANGUAGES (DML)

Methods to store and retrieve information from a database.

**Procedural:**
The query specifies the (high-level) strategy to find the desired result based on sets / bags.

**Non-Procedural (Declarative):**
The query specifies only what data is wanted and not how to find it.

← Relational Algebra
← Relational Calculus
RELATIONAL ALGEBRA

Fundamental operations to retrieve and manipulate tuples in a relation.
→ Based on set algebra (unordered lists with no duplicates).

Each operator takes one or more relations as its inputs and outputs a new relation.
→ We can “chain” operators together to create more complex operations.

σ Select
π Projection
∪ Union
∩ Intersection
− Difference
× Product
⋈ Join
Choose a subset of the tuples from a relation that satisfies a selection predicate.

→ Predicate acts as a filter to retain only tuples that fulfill its qualifying requirement.

→ Can combine multiple predicates using conjunctions / disjunctions.

Syntax: \( \sigma_{\text{predicate}}(R) \)
RELATIONAL ALGEBRA: PROJECTION

Generate a relation with tuples that contains only the specified attributes.
→ Rearrange attributes’ ordering.
→ Remove unwanted attributes.
→ Manipulate values to create derived attributes.

Syntax: $\Pi_{A_1, A_2, \ldots, A_n}(R)$
Generate a relation that contains all tuples that appear in either only one or both input relations.

Syntax: \[ (R \cup S) \]

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
</tr>
</tbody>
</table>

\[ R(a_{id},b_{id}) \]

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>103</td>
</tr>
<tr>
<td>a4</td>
<td>104</td>
</tr>
<tr>
<td>a5</td>
<td>105</td>
</tr>
</tbody>
</table>

\[ S(a_{id},b_{id}) \]

\[ (\text{SELECT} \ast \text{FROM} \ R) \cup (\text{SELECT} \ast \text{FROM} \ S); \]
RELATIONAL ALGEBRA: INTERSECTION

Generate a relation that contains only the tuples that appear in both of the input relations.

Syntax: \((R \cap S)\)

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
</tr>
</tbody>
</table>

\[ R(a_{id},b_{id}) \]

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>103</td>
</tr>
<tr>
<td>a4</td>
<td>104</td>
</tr>
<tr>
<td>a5</td>
<td>105</td>
</tr>
</tbody>
</table>

\[ S(a_{id},b_{id}) \]

\[ (R \cap S) \]

\[ (\text{SELECT} \ast \text{FROM } R) \text{ INTERSECT (SELECT} \ast \text{FROM } S); \]
RELATIONAL ALGEBRA: DIFFERENCE

Generate a relation that contains only the tuples that appear in the first and not the second of the input relations.

Syntax: \((R - S)\)

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
</tr>
</tbody>
</table>

(R)

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
</tbody>
</table>

(S)

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
</tr>
<tr>
<td>a4</td>
<td>104</td>
</tr>
<tr>
<td>a5</td>
<td>105</td>
</tr>
</tbody>
</table>

(R - S)

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
</tbody>
</table>
RELATIONAL ALGEBRA: PRODUCT

Generate a relation that contains all possible combinations of tuples from the input relations.

Syntax: \((R \times S)\)

**SELECT * FROM R CROSS JOIN S;**

**SELECT * FROM R, S;**
RELATIONAL ALGEBRA: JOIN

Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

Syntax: \( (R \bowtie S) \)

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
</tr>
</tbody>
</table>

\[ R(a_{id}, b_{id}) \]

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>103</td>
<td>XXX</td>
</tr>
<tr>
<td>a4</td>
<td>104</td>
<td>YYY</td>
</tr>
<tr>
<td>a5</td>
<td>105</td>
<td>ZZZ</td>
</tr>
</tbody>
</table>
Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

**Syntax:** $(R \bowtie S)$

### Example

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
<th>a_id</th>
<th>b_id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
<td>a3</td>
<td>103</td>
<td>XXX</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Relations

**$R(a_{id},b_{id})$**

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
</tr>
</tbody>
</table>

**$S(a_{id},b_{id},val)$**

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>103</td>
<td>XXX</td>
</tr>
<tr>
<td>a4</td>
<td>104</td>
<td>YYY</td>
</tr>
<tr>
<td>a5</td>
<td>105</td>
<td>ZZZ</td>
</tr>
</tbody>
</table>

$(R \bowtie S)$

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>103</td>
<td>XXX</td>
</tr>
</tbody>
</table>
Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

Syntax: \((R \bowtie S)\)

**RELATIONAL ALGEBRA: JOIN**

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>103</td>
<td>XXX</td>
</tr>
<tr>
<td>a4</td>
<td>104</td>
<td>YYY</td>
</tr>
<tr>
<td>a5</td>
<td>105</td>
<td>ZZZ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a_id</th>
<th>b_id</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>103</td>
<td>XXX</td>
</tr>
<tr>
<td>a4</td>
<td>104</td>
<td>YYY</td>
</tr>
<tr>
<td>a5</td>
<td>105</td>
<td>ZZZ</td>
</tr>
</tbody>
</table>

\(R(a_{id},b_{id})\) \(S(a_{id},b_{id},val)\)
Generate a relation that contains all tuples that are a combination of two tuples (one from each input relation) with a common value(s) for one or more attributes.

**Syntax:** \((R \bowtie S)\)

**RELATIONAL ALGEBRA: JOIN**

\[
\begin{array}{|c|c|} \hline
a\_id & b\_id \\
\hline
a1 & 101 \\
a2 & 102 \\
a3 & 103 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|} \hline
a\_id & b\_id & val \\
\hline
a3 & 103 & XXX \\
a4 & 104 & YYY \\
a5 & 105 & ZZZ \\
\hline
\end{array}
\]

**Example:**

1. \(R(a\_id,b\_id) \bowtie S(a\_id,b\_id,\text{val})\)
2. \(\text{SELECT * FROM } R \text{ NATURAL JOIN } S;\)
3. \(\text{SELECT * FROM } R \text{ JOIN } S \text{ USING } (a\_id, b\_id);\)
4. \(\text{SELECT * FROM } R \text{ JOIN } S \text{ ON } R.a\_id = S.a\_id \text{ AND } R.b\_id = S.b\_id;\)
RELATIONAL ALGEBRA: EXTRA OPERATORS

Rename ($\rho$)
Assignment ($R \leftarrow S$)
Duplicate Elimination ($\delta$)
Aggregation ($\gamma$)
Sorting ($\tau$)
Division ($R \div S$)
Relational algebra defines an ordering of the high-level steps of how to compute a query.

→ Example: $\sigma_{b\_id=102}(R \bowtie S)$ vs. $(R \bowtie (\sigma_{b\_id=102}(S)))$

A better approach is to state the high-level answer that you want the DBMS to compute.

→ Example: Retrieve the joined tuples from $R$ and $S$ where $b\_id$ equals 102.
The relational model is independent of any query language implementation.

**SQL** is the *de facto* standard (many dialects).

```python
for line in file.readlines():
    record = parse(line)
    if record[0] == "GZA":
        print(int(record[1]))

SELECT year FROM artists
WHERE name = 'GZA';
```
DATA MODELS

Relational
Key/Value
Graph

Document / XML / Object ← Leading Alternative
Wide-Column / Column-family
Array / Matrix / Vectors ← Hot these days
Hierarchical
Network
Multi-Value
A collection of record documents containing a hierarchy of named field/value pairs.

→ A field’s value can be either a scalar type, an array of values, or another document.
→ Modern implementations use JSON. Older systems use XML or custom object representations.

Avoid “relational-object impedance mismatch” by tightly coupling objects and database.
DOCUMENT DATA MODEL

```
Artist
  └── ArtistAlbum
    └── Album

R₁(id, ...)
R₂(artist_id, album_id)
R₃(id, ...)
```
DOCUMENT DATA MODEL

Application Code

class Artist {
    int id;
    String name;
    int year;
    Album albums[];
}
class Album {
    int id;
    String name;
    int year;
}

{  
    "name": "GZA",
    "year": 1990,
    "albums": [
        {
            "name": "Liquid Swords",
            "year": 1995
        },
        {
            "name": "Beneath the Surface",
            "year": 1999
        }
    ]
}
VECTOR DATA MODEL

One-dimensional arrays used for nearest-neighbor search (exact or approximate).

→ Used for semantic search on embeddings generated by ML-trained transformer models (think ChatGPT).

→ Native integration with modern ML tools and APIs (e.g., LangChain, OpenAI).

At their core, these systems use specialized indexes to perform NN searches quickly.
VECTOR DATA MODEL

\[ \text{Vector Index} \]

- HNSW
- IVFFlat
- MetaFaiss
- SpotifyAnnoy

\[ \text{Query} \]

Find albums similar to "Liquid Swords"

\[ \text{Embeddings} \]

- \( \text{Id}_1 \rightarrow [0.32, 0.78, 0.30, \ldots] \)
- \( \text{Id}_2 \rightarrow [0.99, 0.19, 0.81, \ldots] \)
- \( \text{Id}_3 \rightarrow [0.01, 0.18, 0.85, \ldots] \)
  \[ \vdots \]

\[ \text{Transformer} \]

\[ \text{RankedList of Ids} \]
CONCLUSION

Databases are ubiquitous.

Relational algebra defines the primitives for processing queries on a relational database.

We will see relational algebra again when we talk about query optimization + execution.
Problem: We want a distributed data structure (many copies) that allows local updates to be made independently and the states eventually converge.
Large communication overhead
Does not allow for disconnected operations
You ❤: 0

Gossip

Another user ❤: Click!

Gossip

Yet another user ❤: Click!

Gossip

You

Another user

Yet another user
Simple example: A “global” counter that each node can independently increment. Nodes can gossip, anytime, and exchange their state. Want each node to “eventually” have the same real global value.
Value() : SUM (myCounter[i] \( i = 0 \ldots \) sizeof(myCounter))

Merge() : myCounter[i] = MAX (myCounter[i], incomingCounter[i]) \( i = 0 \ldots \) sizeof(myCounter)

Magic: Merge() function
\rightarrow Commutative
\rightarrow Associative
\rightarrow Idempotent
Problem: We want a distributed data structure (many copies) that allows local updates to be made independently and the states *eventually converge*.

For P0 the data structure is a set.

**Homework #1** is due Jan 28\textsuperscript{th} @ 11:59pm.
Modern SQL

Make sure you understand basic SQL before the lecture.