Lecture #01

Relational Model & Algebra
SingleStore
**COURSE LOGISTICS**

Course Policies + Schedule: [Course Web Page](#)
Discussion + Announcements: [Piazza](#)
Homeworks + Projects: [Gradescope](#)
Final Grades: [Canvas](#)

Non-CMU students can complete all assignments using [Gradescope](#) (Code: **KK5DVJ**).

→ Do **not** post your solutions on Github.
→ Do **not** email instructors / TAs for help.
→ Discord Channel: [https://discord.gg/YF7dMCg](#)
→ Somebody needs to finish Andy's [Wikipedia](#) article.
Pavlo was born and raised in the streets of Baltimore, MD. After completing bachelor's and master's degrees from Rochester Institute of Technology and Brown University, he completed his Ph.D. from Brown University under Stan Zdonik and Mike Stonebraker.

→ Do not post your solutions on Github.
→ Do not email instructors / TAs for help.
→ Discord Channel: https://discord.gg/YF7dMCg
→ Somebody needs to finish Andy's Wikipedia article.
LECTURE RULES

Do interrupt us for the following reasons:
→ We are speaking too fast.
→ You don't understand what we are talking about.
→ You have a database-related question.

Do **not** interrupt us for the following reasons:
→ Whether you can use the bathroom.
→ Questions about blockchains.

We will **not** answer questions about the lecture immediately after class.
TODAY’S AGENDA

Database Systems Background
Relational Model
Relational Algebra
Alternative Data Models
Databases
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.

Artist(name, year, country)

"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"GZA",1990,"USA"

Album(name, artist, year)

"Enter the Wu-Tang","Wu-Tang Clan",1993
"St.Ides Mix Tape","Wu-Tang Clan",1994
"Liquid Swords","GZA",1990
Example: Get the year that GZA went solo.

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

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

- "Wu-Tang Clan", 1992, "USA"
- "Notorious BIG", 1992, "USA"
- "GZA", 1990, "USA"
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?
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.
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
- Hierarchical
- Network
- Multi-Value

← Machine Learning
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

← This Course
EARLY DBMSs

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.
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

Before users of large data banks must be protected from having to know how the data is organized in the machine (the external representation). A prompting service which supplies such information to a satisfactory degree of precision. Activities of users and human use programs should remain unaffected when the internal representation of the data is changed and even when any aspects of the external representation are changed. Changes in data representation will often be carried on as a result of changes in query, update, or report traffic and natural growth in the types of stored information. Existing, non-relational, information systems provide users with machine-readable files or highly general network stores of the data. In Section 1, the design of the model is discussed. A model based on various relations, a normal form for data base relations, and the concept of a universal data sublanguage is introduced. In Section 2, certain operators on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user’s model.

1. Relational Model and Normal Form

1.1 Introduction

This paper is concerned with the applications of the relational theory to systems which provide shared access to large banks of sequential data. Although a paper by Childs [2] the principal application of relations to data systems has been to deductive query-analyzing systems. Jones and Mann [3] provide numerous references to work in this area.

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

1.2 Data Dependencies in Primary Systems

The provision of a data description table is normally a prerequisite to the goal of data independence [5, 6, 7]. Such tables facilitate classifying certain changes of the data representation stored in a data bank. However, the variety of data representation which can be changed without logically impairing some application programs is still quite limited. Further, the model of data which users need is still difficult with programmable representations, particularly in regard to the representation of relations of data (as opposed to individual items). Three of the principal kinds of data independence which still need to be resolved are: ordering dependency, interdependence, and source path dependence. In some systems these kinds of data 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 by sorting to order for quick access, others for grouping to participate in one ordering only, others permitting each element to participate in several orders. This problem occurs whenever a data bank is to be shared by any combination of systems whose existing elements either require or permit data elements to be stored in at least one total ordering which is closely associated with the specification-determining conditions 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 is a rearrangement of) the

The relationship of (or models of) data described in Section 1 appears to be superior in several respects to the graph or network model [5, 6] presently in vogue for non-relational systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a data bank that is hierarchically organized, languages which will yield maximum independence between the hand and machine representation and organization of data.
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.
RELATIONAL MODEL

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).

<table>
<thead>
<tr>
<th>Artist(name, year, country)</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>Wu-Tang Clan</td>
</tr>
<tr>
<td>Notorious BIG</td>
</tr>
<tr>
<td>GZA</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)
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

<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

<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>
## Relations Model: Foreign Keys

**Artist**

<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**

<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**

<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

<table>
<thead>
<tr>
<th>Artist(id, name, year, country)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>102</td>
</tr>
<tr>
<td>103</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ArtistAlbum(artist_id, album_id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>artist_id</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>103</td>
</tr>
<tr>
<td>102</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Album(id, name, year)</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>33</td>
</tr>
</tbody>
</table>
RELATIONAL MODEL: CONSTRAINTS

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).

<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 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
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.
RELATIONAL ALGEBRA: SELECT

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: $\sigma_{\text{predicate}}(R)$

<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>a2</td>
<td>103</td>
</tr>
<tr>
<td>a3</td>
<td>104</td>
</tr>
</tbody>
</table>

$\sigma_{\text{a_id='a2'}}(R)$

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

$\sigma_{\text{a_id='a2' AND b_id>102}}(R)$

SELECT * FROM R
WHERE a_id='a2' AND b_id>102
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: \((R \cup S)\)
Generate a relation that contains only the tuples that appear in both of the input relations.

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

\[
\begin{array}{cc}
R(a_{id}, b_{id}) & S(a_{id}, b_{id}) \\
\hline
a_{id} & b_{id} & a_{id} & b_{id} \\
101 & 103 & a3 & 103 \\
102 & 104 & a4 & 104 \\
103 & 105 & a5 & 105 \\
\end{array}
\]

\[
(\text{SELECT} \ast \text{FROM} \ R) \ \text{EXCEPT} \ (\text{SELECT} \ast \text{FROM} \ S);
\]
### 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;
```

---

<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>
</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>

---

<table>
<thead>
<tr>
<th>R.a_id</th>
<th>R.b_id</th>
<th>S.a_id</th>
<th>S.b_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>101</td>
<td>a3</td>
<td>103</td>
</tr>
<tr>
<td>a1</td>
<td>101</td>
<td>a4</td>
<td>104</td>
</tr>
<tr>
<td>a1</td>
<td>101</td>
<td>a5</td>
<td>105</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
<td>a3</td>
<td>103</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
<td>a4</td>
<td>104</td>
</tr>
<tr>
<td>a2</td>
<td>102</td>
<td>a5</td>
<td>105</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
<td>a3</td>
<td>103</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
<td>a4</td>
<td>104</td>
</tr>
<tr>
<td>a3</td>
<td>103</td>
<td>a5</td>
<td>105</td>
</tr>
</tbody>
</table>
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: \((R \bowtie S)\)

\[
\begin{array}{|c|c|}
\hline
\text{a\_id} & \text{b\_id} \\
\hline
\text{a1} & 101 \\
\text{a2} & 102 \\
\text{a3} & 103 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{a\_id} & \text{b\_id} & \text{val} \\
\hline
\text{a3} & 103 & \text{XXX} \\
\text{a4} & 104 & \text{YYY} \\
\text{a5} & 105 & \text{ZZZ} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{R.a\_id} & \text{R.b\_id} \\
\hline
\text{a3} & 103 \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{S.a\_id} & \text{S.b\_id} & \text{S.val} \\
\hline
\text{a3} & 103 & \text{XXX} \\
\hline
\end{array}
\]

\[(R \bowtie S)\]
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: \((R \bowtie S)\)

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

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

\[
\begin{array}{|c|c|}
\hline
\text{a\_id} & \text{b\_id} \\
\hline
\text{a3} & 103 \\
\hline
\end{array}
\]

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

SELECT * FROM R NATURAL JOIN S;

SELECT * FROM R JOIN S USING (a_id, b_id);

SELECT * FROM R JOIN S
  ON R.a_id = S.a_id AND R.b_id = S.b_id;
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]))
```

```sql
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 ← Current Hotness
Hierarchical
Network
Multi-Value
DOCUMENT DATA MODEL

A collection of record documents containing a hierarchy of named field/value pairs.
→ A field's value can 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

\[ R_1(id,...) \]

\[ R_2(artist\_id,album\_id) \]

\[ R_3(id,...) \]
DOCUMENT DATA MODEL

Artist

→ $R_1(id,\ldots)$

Artist → Album

★

→ $R_2(artist\_id,album\_id)$

★

→ $R_3(id,\ldots)$

Album
**DOCUMENT DATA MODEL**

**Application Code**

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

class Album {
    int id;
    String name;
    int year;
}
```
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

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.

- Pinecone
- Weaviate
- milvus
- drant
- marqo
- LanceDB
- {feature}form
VECTOR DATA MODEL

**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>

**Embeddings**

- `Id1 → [0.32, 0.78, 0.30, ...]`
- `Id2 → [0.99, 0.19, 0.81, ...]`
- `Id3 → [0.01, 0.18, 0.85, ...]`

**Vector Index**

- HNSW, IVFFlat
- Meta Faiss, Spotify Annoy
VECTOR DATA MODEL

### Album

<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>

### Embeddings

- Id1 → [0.32, 0.78, 0.30, ...]
- Id2 → [0.99, 0.19, 0.81, ...]
- Id3 → [0.01, 0.18, 0.85, ...]
- Id4 → [0.02, 0.10, 0.24, ...]

### Query

Find albums similar to "Liquid Swords"
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.
Modern SQL
→ Make sure you understand basic SQL before the lecture.