Course Intro & Relational Model
TODAY’S AGENDA

Wait List
Overview
Course Logistics
Relational Model
Relational Algebra
I do **not** control the wait list.

Students will be moved off the wait list (based on position) as new spots become available.

If you are not currently enrolled, the likelihood that you will get in is unfortunately very low.
Please interrupt me at any time if:
→ I am speaking too fast.
→ You don't understand what I am talking about.
→ You have a database-related question.
This course is about the design/implementation of database management systems (DBMSs).

This is not a course about how to use a DBMS to build applications or how to administer a DBMS. → See CMU 95-703 (Heinz College)

Database Applications (15-415/615) is not offered this semester.
COURSE OUTLINE

Relational Databases
Storage
Execution
Concurrency Control
Recovery
Distributed Databases
Potpourri
COURSE LOGISTICS

Course Policies + Schedule:
→ Refer to course web page.

Academic Honesty:
→ Refer to CMU policy page.
→ If you're not sure, ask one of the instructors.
→ Generally, don't be stupid.

All discussion + announcements will be on Piazza.
Database System Concepts
7th Edition
Silberschatz, Korth, & Sudarshan

We will also provide lecture notes that cover topics not found in textbook.
COURSE RUBRIC

Homeworks (15%)
Projects (45%)
Midterm Exam (20%)
Final Exam (20%)
Five homework assignments throughout the semester.

First homework is a SQL assignment. The rest will be pencil-and-paper assignments.

All homework should be done individually.
You will build your own database engine from scratch over the course of the semester.

Each project builds on the previous one.

We will **not** teach you how to write/debug C++17. It is a prerequisite for the course.

You must complete [Project #0](#) before Sept 13th.
All projects will use the CMU DB Group **BusTub** academic DBMS.

Architecture:
- Disk-based Storage
- Volcano-style Query Processing
- Pluggable APIs
- Currently does not support SQL
LATE POLICY

You will lose 10% of the points for a project or homework for every 24 hours it is late.

You have a total of four late days to be used for projects only.

We will grant no-penalty extensions due to extreme circumstances (e.g., medical emergencies).
→ If something comes up, please contact the instructors as soon as possible.
PLAGIARISM WARNING

The homework and projects must be your own original work. They are **not** group assignments.

You may **not** copy source code from other people or the web.

Plagiarism will **not** be tolerated.

See [CMU's Policy on Academic Integrity](#) for additional information.
We are still waiting on clarification from the university about in-person vs. remote office hours.

As soon as we know more, we will make an announcement and update the website.

If you need to contact us sooner, please send an email or reach out on Piazza.
Vaccination Database Tech Talks
→ Mondays @ 4:30pm (starting on 9/13)
→ https://db.cs.cmu.edu/seminar2021-dose2
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.
Create a database that models a digital music store.

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

- "Wu-Tang Clan", 1992, "USA"
- "Notorious BIG", 1992, "USA"
- "Ice Cube", 1989, "USA"

**Album**(name, artist, year)

- "Enter the Wu-Tang", "Wu-Tang Clan", 1993
- "St. Ides Mix Tape", "Wu-Tang Clan", 1994
- "AmeriKKKa's Most Wanted", "Ice Cube", 1990
Example: Get the year that Ice Cube went solo.

<table>
<thead>
<tr>
<th>Artist</th>
<th>name</th>
<th>year</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Wu-Tang Clan&quot;</td>
<td>1992</td>
<td>&quot;USA&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Notorious BIG&quot;</td>
<td>1992</td>
<td>&quot;USA&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Ice Cube&quot;</td>
<td>1989</td>
<td>&quot;USA&quot;</td>
<td></td>
</tr>
</tbody>
</table>
Example: Get the year that Ice Cube went solo.

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

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

"Wu-Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"Ice Cube",1989,"USA"
FLAT FILES: DATA INTEGRITY
FLAT FILES: DATA INTEGRITY

How do we ensure that the artist is the same for each album entry?
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?
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?
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
FLAT FILES: IMPLEMENTATION

How do you find a particular record?
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?
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 two threads try to write to the same file at the same time?
FLAT FILES: DURABILITY
FLAT FILES: DURABILITY

What if the machine crashes while our program is updating a record?
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 is designed to allow the definition, creation, querying, update, and administration of databases.
Database applications were difficult to build and maintain.

Tight coupling between logical and physical layers.

You have to (roughly) know what queries your app would execute before you deployed the database.
Database applications were difficult to build and maintain.

Tight coupling between logical and physical layers.

You have to (roughly) know what queries your app would execute before you deployed the database.
Database applications were difficult to build and maintain.

Tight coupling between logical and physical data structures.

You had to (roughly) know what queries your app would execute before deploying the database.

Edgar F. Codd
Database applications were difficult to build and maintain. Tight coupling between logical and physical layers. You have to (roughly) know what queries your app would execute before you deployed the database.
Proposed in 1970 by Ted Codd.

Database abstraction to avoid this maintenance:
→ Store database in simple data structures.
→ Access data through high-level language, DBMS figures out best strategy.
→ Physical storage left up to the DBMS implementation.
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 MODEL

Relational
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network
Multi-Value
DATA MODEL

Relational ← Most DBMSs
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network
Multi-Value
DATA MODEL

Relational
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network
Multi-Value

← NoSQL
DATA MODEL

Relational
Key/Value
Graph
Document
Column-family
Array / Matrix ← Machine Learning
Hierarchical
Network
Multi-Value
DATA MODEL

Relational
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network
Multi-Value

← Obsolete / Legacy / Rare
DATA MODEL

Relational
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network
Multi-Value
DATA MODEL

Relational ← This Course
Key/Value
Graph
Document
Column-family
Array / Matrix
Hierarchical
Network
Multi-Value
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.
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.

### 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>Ice Cube</td>
<td>1989</td>
<td>USA</td>
</tr>
</tbody>
</table>
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.

<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>Ice Cube</td>
<td>1989</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.

Auto-generation of unique integer primary keys:
- → **SEQUENCE** (SQL:2003)
- → **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.

Auto-generation of unique integer primary keys:
→ **SEQUENCE** (SQL:2003)
→ **AUTO_INCREMENT** (MySQL)
A foreign key specifies that an attribute from one relation has to map 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>123</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>456</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>789</td>
<td>Ice Cube</td>
<td>1989</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>123</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>AmeriKKKa's Most Wanted</td>
<td>789</td>
<td>1990</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>123</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>456</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>789</td>
<td>Ice Cube</td>
<td>1989</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>123</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>AmeriKKKa's Most Wanted</td>
<td>789</td>
<td>1990</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>123</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>456</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>789</td>
<td>Ice Cube</td>
<td>1989</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>123</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>AmeriKKKa's Most Wanted</td>
<td>789</td>
<td>1990</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>123</td>
<td>11</td>
</tr>
<tr>
<td>123</td>
<td>22</td>
</tr>
<tr>
<td>789</td>
<td>22</td>
</tr>
<tr>
<td>456</td>
<td>22</td>
</tr>
</tbody>
</table>
## 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>123</td>
<td>Wu-Tang Clan</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>456</td>
<td>Notorious BIG</td>
<td>1992</td>
<td>USA</td>
</tr>
<tr>
<td>789</td>
<td>Ice Cube</td>
<td>1989</td>
<td>USA</td>
</tr>
</tbody>
</table>

### 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>AmeriKKA's Most Wanted</td>
<td>1990</td>
</tr>
</tbody>
</table>

### ArtistAlbum

<table>
<thead>
<tr>
<th>artist_id</th>
<th>album_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>11</td>
</tr>
<tr>
<td>123</td>
<td>22</td>
</tr>
<tr>
<td>789</td>
<td>22</td>
</tr>
<tr>
<td>456</td>
<td>22</td>
</tr>
</tbody>
</table>
## RELATIONAL MODEL: FOREIGN KEYS

The image shows a relational model with three tables: `Artist`, `ArtistAlbum`, and `Album`. The `Artist` table includes columns for `id`, `name`, `year`, and `country`. The `ArtistAlbum` table has columns for `artist_id` and `album_id`. The `Album` table contains columns for `id`, `name`, and `year`. The relationships are visualized with arrows indicating foreign keys:

- From `Artist` to `ArtistAlbum` via `id` (artist_id)
- From `ArtistAlbum` to `Album` via `album_id` (artist_id)
- From `ArtistAlbum` to `Album` via `album_id` (album_id)

### Artist Table
- **id**: 123
- **name**: Wu-Tang Clan
- **year**: 1992
- **country**: USA
- **id**: 456
- **name**: Notorious BIG
- **year**: 1992
- **country**: USA
- **id**: 789
- **name**: Ice Cube
- **year**: 1989
- **country**: USA

### ArtistAlbum Table
- **artist_id**: 123
- **album_id**: 11
- **artist_id**: 123
- **album_id**: 22
- **artist_id**: 789
- **album_id**: 22
- **artist_id**: 456
- **album_id**: 22

### Album Table
- **id**: 11
- **name**: Enter the Wu-Tang
- **year**: 1993
- **id**: 22
- **name**: St.Ides Mix Tape
- **year**: 1994
- **id**: 33
- **name**: AmeriKKKa's Most Wanted
- **year**: 1990
Methods to store and retrieve information from a database.

**Procedural:**
→ The query specifies the (high-level) strategy the DBMS should use to find the desired result.

**Non-Procedural (Declarative):**
→ The query specifies only what data is wanted and not how to find it.
Methods to store and retrieve information from a database.

**Procedural:**

→ The query specifies the (high-level) strategy the DBMS should use to find the desired result.

**Non-Procedural (Declarative):**

→ The query specifies only what data is wanted and not how to find it.
Methods to store and retrieve information from a database.

**Procedural:**
→ The query specifies the (high-level) strategy the DBMS should use to find the desired result.

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

↔ Relational Algebra
↔ Relational Calculus
Methods to store and retrieve information from a database.

**Procedural:**
→ The query specifies the (high-level) strategy the DBMS should use to find the desired result.

**Non-Procedural (Declarative):**
→ The query specifies only what data is wanted and not how to find it.
Fundamental operations to retrieve and manipulate tuples in a relation. → Based on set algebra.

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

\[
\begin{array}{|c|c|}
\hline
\text{a_id} & \text{b_id} \\
\hline
\text{a1} & 101 \\
\text{a2} & 102 \\
\text{a2} & 103 \\
\text{a3} & 104 \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|}
\hline
\text{a_id} & \text{b_id} \\
\hline
\text{a2} & 102 \\
\text{a2} & 103 \\
\hline
\end{array}
\]

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

\[
\begin{array}{|c|c|}
\hline
\text{a_id} & \text{b_id} \\
\hline
\text{a2} & 103 \\
\hline
\end{array}
\]

SELECT * FROM R WHERE a_id='a2' AND b_id>102;
RELATIONAL ALGEBRA: PROJECTION

Generate a relation with tuples that contains only the specified attributes.
→ Can rearrange attributes’ ordering.
→ Can manipulate the values.

Syntax: $\pi_{A_1,A_2,...,A_n}(R)$
Generate a relation with tuples that contains only the specified attributes.
→ Can rearrange attributes’ ordering.
→ Can manipulate the values.

**Syntax:** \( \Pi_{A_1,A_2,\ldots,A_n}(R) \)

**Example:**

\[ R(a\_id,b\_id) \]

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

\( \Pi_{b\_id=100,a\_id}(\sigma_{a\_id='a2'}(R)) \)

<table>
<thead>
<tr>
<th>( b_id=100 )</th>
<th>( a_id )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>a2</td>
</tr>
<tr>
<td>3</td>
<td>a2</td>
</tr>
</tbody>
</table>
RELATIONAL ALGEBRA: PROJECTION

Generate a relation with tuples that contains only the specified attributes.
→ Can rearrange attributes’ ordering.
→ Can manipulate the values.

Syntax: \( \Pi_{A_1,A_2,\ldots,A_n}(R) \)

\[
\begin{array}{|c|c|}
\hline
\text{a_id} & \text{b_id} \\
\hline
\text{a1} & 101 \\
\text{a2} & 102 \\
\text{a2} & 103 \\
a3 & 104 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{b_id-100} & \text{a_id} \\
\hline
2 & \text{a2} \\
3 & \text{a2} \\
\hline
\end{array}
\]

\[
\text{SELECT b_id-100, a_id FROM R WHERE a_id = 'a2';}
\]
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 all tuples that appear in either only one or both input relations.

Syntax: \((R \cup S)\)
RELATIONAL ALGEBRA: UNION

Generate a relation that contains all tuples that appear in either only one or both input relations.

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

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

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

\[
\begin{array}{|c|c|}
\hline
\text{(SELECT * FROM R)} & \text{UNION ALL} \\
\text{(SELECT * FROM S)}; \\
\hline
a1 & 101 \\
a2 & 102 \\
a3 & 103 \\
a3 & 103 \\
a4 & 104 \\
a5 & 105 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{(R \cup S)} & \\
\hline
a1 & 101 \\
a2 & 102 \\
a3 & 103 \\
a3 & 103 \\
a4 & 104 \\
a5 & 105 \\
\hline
\end{array}
\]
Generate a relation that contains only the tuples that appear in both of the input relations.

**Syntax:** \((R \cap S)\)
Generate a relation that contains only the tuples that appear in both of the input relations.

**Syntax:** \((R \cap S)\)
Generate a relation that contains only the tuples that appear in the first and not the second of the input relations.

Syntax: \((R - S)\)
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>

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

<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>

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

$(R - S)$
Generate a relation that contains only the tuples that appear in the first and not the second of the input relations.

Syntax: \((R - S)\)
RELATIONAL ALGEBRA: PRODUCT

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

Syntax: \((R \times S)\)
Generate a relation that contains all possible combinations of tuples from the input relations.

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

<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>
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: \((R \bowtie S)\)
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: EXTRA OPERATORS

Rename ($\rho$)
Assignment ($R \leftarrow S$)
Duplicate Elimination ($\delta$)
Aggregation ($\gamma$)
Sorting ($\tau$)
Division ($R \div S$)
Observation

Relational algebra still defines the high-level steps of how to compute a query.

\[ \sigma_{b\_id=102}(R \bowtie S) \text{ vs. } (R \bowtie (\sigma_{b\_id=102}(S))) \]
Relational algebra still defines the high-level steps of how to compute a query.
→ $\sigma_{b\_id=102}(R \bowtie S)$ 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.
→ 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).
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] == "Ice Cube":
        print(int(record[1]))
```
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] == "Ice Cube":
        print(int(record[1]))
```

```sql
SELECT year FROM artists
WHERE name = 'Ice Cube';
```
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.
NEXT CLASS

Crash Course on SQL