01 Course Intro & Relational Model
Otter King
TODAY’S AGENDA

Wait List
Overview
Course Logistics
Relational Model
Relational Algebra
WAIT LIST

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.
LECTURE RULES

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.
COURSE OVERVIEW

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](https://example.edu/course) (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%)
HOMEWORKS

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.
PROJECTS

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.
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.
OFFICE HOURS

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](https://db.cs.cmu.edu/seminar2021-dose2)
Carnegie Mellon University

VACCINATION
Database Talks
SECOND DOSE

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.
DATABASE EXAMPLE

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

`Artist(name, year, country)`

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

```python
for line in file.readlines():
    record = parse(line)
    if record[0] == "Ice Cube":
        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 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 is designed to allow the definition, creation, querying, update, and administration of databases.
EARLY DBMSs

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.

Edgar F. Codd
A Relational Model of Data for Large Shared Databases

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

Abstract: Users of large data banks must be protected from the necessity of knowing how the data are organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory substitute, Activites of users of terminals and most applications programs would be made easier if changes in the internal representation of data were transparent to the users. This requires that the operations supported by the system be defined independently of the representation of the data. This can be achieved by the use of a relational model, in which the concept of a data schema is separated from the physical implementation of the schema.

The relational model is based on the theory of abstract algebra, particularly on the theory of relations. It is a formalism for specifying and manipulating data, and it is independent of the physical implementation of the data. This independence is achieved by the use of a relational model, in which the concept of a data schema is separated from the physical implementation of the schema.

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 bodies of structured data. Examples of such systems are large sets of records, large sets of data, large sets of documents, large sets of transactions, and large sets of data tables. This paper provides an introduction to these systems.

In contrast, the problems treated here are those of data independence—such independence being independent of the underlying data structure and terminal activities. The system has been designed to accommodate data and change in data representation and to provide certain kinds of data manipulation which are expected to become commonplace even in unstructured systems.
RELATIONAL MODEL

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.

Edgar F. Codd
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 ← 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.

<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'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>

### 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>

### 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>AmeriK KK'a's Most Wanted</td>
<td>1990</td>
</tr>
</tbody>
</table>
DATA MANIPULATION LANGUAGES (DML)

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**

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.

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

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

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

<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>

SELECT b_{id}=100, a_{id} FROM R WHERE a_{id} = 'a2';
## RELATIONAL ALGEBRA: UNION

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

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

### Example

<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>

\[(R \cup S) = \begin{array}{l}
(a_1, b_{101}) \\
(a_2, b_{102}) \\
(a_3, b_{103}) \\
(a_4, b_{104}) \\
(a_5, b_{105})
\end{array}\]

\((R \cup S) = \begin{array}{l}
(a_1, b_{101}) \\
(a_2, b_{102}) \\
(a_3, b_{103}) \\
(a_4, b_{104}) \\
(a_5, b_{105})
\end{array}\)

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

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

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

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

\[
\begin{array}{|c|c|}
\hline
a_id & b_id \\
\hline
a3 & 103 \\
a4 & 104 \\
a5 & 105 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
a_id & b_id \\
\hline
a3 & 103 \\
\hline
\end{array}
\]

\[
\text{(SELECT * FROM } R) \quad \text{INTERSECT} \quad \text{(SELECT * FROM } S)\text{;}
\]
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: \((R - S)\)

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

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

\((R - S)\)

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

\[
\text{(SELECT * FROM R) EXCEPT (SELECT * FROM S);}
\]
RELATIONAL ALGEBRA: PRODUCT

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

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

\[
\begin{array}{|c|c|}
\hline
R(a_{id}, b_{id}) & S(a_{id}, b_{id}) \\
\hline
a_{id} & b_{id} & a_{id} & b_{id} \\
\hline
a1 & 101 & a3 & 103 \\
a2 & 102 & a4 & 104 \\
a3 & 103 & a5 & 105 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
R.a_{id} & R.b_{id} & S.a_{id} & S.b_{id} \\
\hline
a1 & 101 & a3 & 103 \\
a1 & 101 & a4 & 104 \\
a1 & 101 & a5 & 105 \\
a2 & 102 & a3 & 103 \\
a2 & 102 & a4 & 104 \\
a2 & 102 & a5 & 105 \\
a3 & 103 & a3 & 103 \\
a3 & 103 & a4 & 104 \\
a3 & 103 & a5 & 105 \\
\hline
\end{array}
\]

SELECT * FROM \(R\) CROSS JOIN \(S\);

SELECT * FROM \(R\), \(S\);
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: \((R \bowtie S)\)

```
SELECT * FROM R NATURAL JOIN S;
```
RELATIONAL ALGEBRA: EXTRA OPERATORS

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

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

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

→ R

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] == "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