Leon Wrinkles
(1946-2018)
#1 – KB

#2 – Databases
TODAY’S AGENDA

Wait List
Overview
Course Logistics
Relational Model
Relational Algebra
WAIT LIST

There are currently 150 people on the waiting list. Max capacity is 100.

We will enroll people from the waiting list in the order that you complete Homework #1.
COURSE OVERVIEW

This course is on the design and implementation of disk-oriented database management systems.

This is **not** a course on how to use a database to build applications or how to administer a database. → See [CMU 95-703](#) (Heinz College)

Database Applications (**15-415/615**) is cancelled 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 the professors.
→ Don’t be stupid.

All discussion + announcements will be on Piazza.
Database System Concepts
6th Edition
Silberschatz, Korth, & Sudarshan

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

Homeworks (15%)
Projects (45%)
Midterm Exam (20%)
Final Exam (20%)
Extra Credit (+10%)
HOMEWORKS

Five homework assignments throughout the semester.

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

All homeworks should be done individually.
PROJECTS

You will build your own storage manager from scratch of the course of the semester. Each project builds on the previous one.

We will not teach you how to write/debug C++11 code.
LATE POLICY

You are allowed 4 slip days for either homeworks or projects.

You lose 25% of an assignment’s points for every 24hrs it is late.

Mark on your submission (1) how many days you are late and (2) how many late days you have left.
The homeworks and projects must be your own 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.
Database Group Meetings
→ Mondays @ 4:30pm (GHC 8102)
→ https://db.cs.cmu.edu

Peloton Developer Meetings
→ Tuesdays @ 12:00pm (GHC 8115)
→ https://pelotondb.io

Database Seminar Series
→ Thursdays @ 12:00pm (CIC 4th Floor)
→ https://db.cs.cmu.edu/seminar2018
Databases
DATABASE

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

Databases are core the 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 store:
→ Information about Artists
→ What Albums those Artists released
FLAT FILE STRAWMAN

Store our database as comma-separated value (CSV) files that we manage in our own code.
→ Use a separate file per entity.
→ The application has to 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.

```
Artist(name, year, country)
"Wu Tang Clan",1992,"USA"
"Notorious BIG",1992,"USA"
"Ice Cube",1989,"USA"
```

```python
for line in file:
    record = parse(line)
    if "Ice Cube" == record[0]:
        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?

How do we store that there are multiple artists on an album?
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?
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 **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 Data Banks

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

Abstract

When users of large data banks must be protected from having to know how the data is actually organized in the machine (the internal representation), a promising service which supplies such information is a satisfactory one. Activity of users at terminals and most applications programs should be independent of the changes in their data representation. Changes in data representation are often referred to as "changes in view". The more important changes are those which affect the performance of data retrieval and updating, and these are often referred to as "changes in schema".

The model based on such a view is a normal form for data base relations, and the concept of a universal data sublanguage is introduced in Section 2. Certain operations are defined on relations (after logical inference) and applied to the problem of redundancy and consistency in the view's model.

1. Relational Model and Normal Form

1.1. Introduction

This paper is concerned with the application of classificatory and relational theory to systems which provide access to large banks of numerical data. Described in a paper by Codd [1], the principal application of relations to data system has been to the design of question-answering systems, based on the model of the data base "database" (in a relational language) as a multiuser, multi-software, multi-purpose, multi-language, multi-relation, multi-value, multi-type, multi-field, multi-record, multi-file, multi-system database. The model is intended for computer systems which are designed to support the relational model, and it is intended that the model be used to design computer systems which are intended to support the relational model.

In contrast, the principal problem today is the interdependence of application programs and terminal activities from the growth in data type and change in data representation—certain kinds of data which are expected to become more elaborate over time in relation databases.

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] presently in vogue for non-relational systems. It provides a means of describing data with its natural structure only that it is without presupposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high-level database language which will yield maximum independence between programs on the one hand and machine representation and organization of databases on the other.

The relational view makes possible the encoding of the schema and logical structure of data (and logical implications of structure) in terms of the data model and an alphabet of key expressions. This enables users to make changes in data representation without changing applications or maintaining different views of the data. It also enables users to change the data model without affecting the applications which are based on it. In other words, the relational view enables users to change the schema without changing the data.
Proposed in 1970 by Ted Codd.

Database abstraction to avoid this maintenance:
→ Store database in simple data structures.
→ Access data through high-level language.
→ Physical storage left up to 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

← Most DBMSs
DATA MODEL

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

← NoSQL
DATA MODEL

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

← Machine Learning
DATA MODEL

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

← Obsolete / Rare
DATA MODEL

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

← This Course
RELATIONAL MODEL

Structure: The definition of relations and their contents.

Integrity: Ensure the database’s contents satisfy constraints.

Manipulation: How to access and modify 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.

**n-ary Relation**

= Table with n columns

<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>
RELATIONAL MODEL: PRIMARY KEYS

A relation’s primary key uniquely identifies a single tuple.

Some DBMSs automatically create an internal primary key if you don't define one.

Auto-generation of unique integer primary keys:

→ **SEQUENCE** (SQL:2003)

→ **AUTO_INCREMENT** (MySQL)

<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>
RELATIONAL MODEL: PRIMARY KEYS

A relation’s primary key uniquely identifies a single tuple.

Some DBMSs automatically create an internal primary key if you don't define one.

Auto-generation of unique integer primary keys:

→ **SEQUENCE** (SQL:2003)
→ **AUTO_INCREMENT** (MySQL)

<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>
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><strong>Enter the Wu Tang</strong></td>
<td>123</td>
<td>1993</td>
</tr>
<tr>
<td>22</td>
<td><strong>St.Ides Mix Tape</strong></td>
<td>???</td>
<td>1994</td>
</tr>
<tr>
<td>33</td>
<td><strong>AmeriKKKa's Most Wanted</strong></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, 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 Table

<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 Table

<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 Table

<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>AmeriKKKa's Most Wanted</td>
<td>1990</td>
</tr>
</tbody>
</table>
How 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:**
→ 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: \( \pi_{A_1, A_2, \ldots, A_n}(R) \)

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

\[
\sigma_{\text{a_id}=\text{a2}}(R)
\]

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

\[
\pi_{\text{b_id}=100, \text{a_id}}(\sigma_{\text{a_id}=\text{a2}}(R))
\]

\[
\text{SELECT } \text{b_id}=100, \text{a_id} \\
\text{FROM } R \text{ WHERE } \text{a_id} = \text{'a2'};
\]
RELATIONAL ALGEBRA: UNION

Generate a relation that contains all tuples that appear in either only one or both of the 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>

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

\((R \cup S)\)

\((\text{SELECT } * \text{ FROM } R) \ \text{UNION} \ \text{SELECT } * \text{ FROM } S;\)
Generate a relation that contains only the tuples that appear in both of the input relations.

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

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

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

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

\[
\text{(SELECT * FROM R)} \ \text{INTERSECT} \ \text{(SELECT * FROM S)};
\]
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>

<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>

\( (\text{SELECT * FROM } R) \text{ EXCEPT } (\text{SELECT * FROM } S) \);
Generate a relation that contains all possible combinations of tuples from the input relations.

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

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

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

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

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

\[(R \bowtie S)\]

\[
\text{SELECT } * \text{ FROM } R \text{ NATURAL JOIN } S;
\]
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.

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

A better approach is just state the high-level query you want

→ 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.

```python
for line in file:
    record = parse(line)
    if "Ice Cube" == record[0]:
        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.