Carnegie Mellon University

# 01 Course Intro & Relational Model

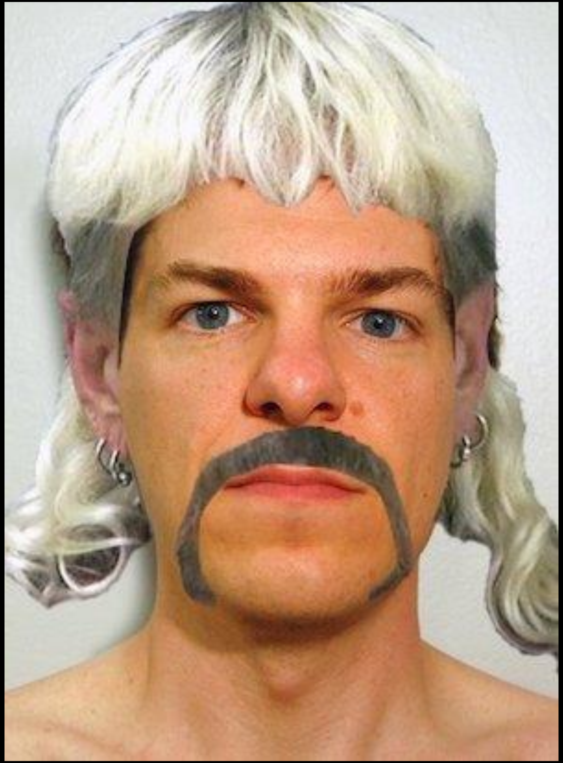
Intro to Database Systems
15-445/15-645
Fall 2021

Andrew Crotty
Computer Science
Carnegie Mellon University

≠

$\neq$

$\neq$



# Otter King

Google
Big Query

# 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 (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.

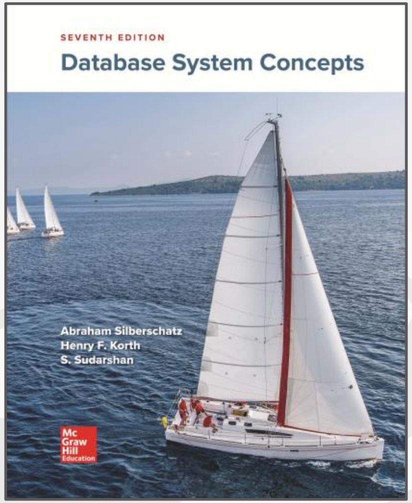
Academic Honesty:
→ Refer to CMU policy page.
→ If you're not sure, ask one of the instructors.
→ Generally, don't be stupid.

All discussion + announcements will be on Piazza.

# TEXTBOOK

**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.

# BUSTUB

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

BusTub

# 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 **<u>four</u>** late days to be used for **<u>projects only</u>**.

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.

# 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.

# DATABASE RESEARCH

**Vaccination Database Tech Talks**

→ Mondays @ 4:30pm (starting on 9/13)

→ https://db.cs.cmu.edu/seminar2021-dose2


rqlite | dbt | Pinecone | zerowatt | [tile]DB

Google | amazon REDSHIFT | fluree | ARROW | bodo.ai

trino | dremio | FIREBOLT

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.

# FLAT FILE STRAWMAN

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
```

# FLAT FILE STRAWMAN

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"
```

# FLAT FILE STRAWMAN

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"
```

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

# FLAT FILES: DATA INTEGRITY

# FLAT FILES: DATA INTEGRITY

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

# 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?

# 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?

# 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

# 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?

# DATABASE MANAGEMENT SYSTEM

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.

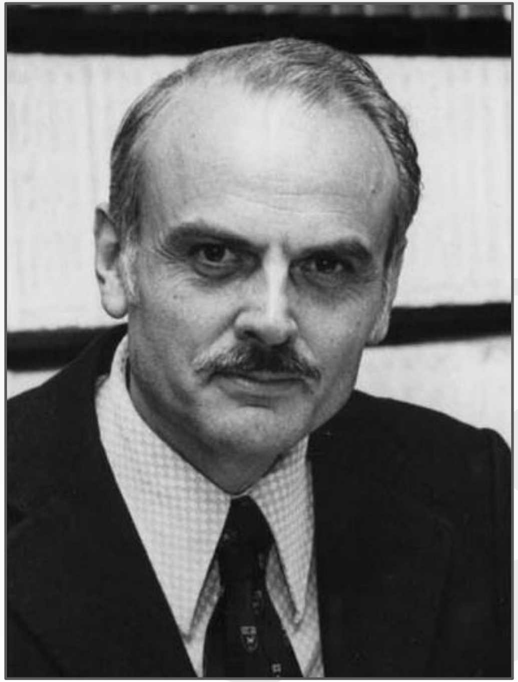
You have to (roughly) know what queries your app would execute before you deployed the database.

# 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.



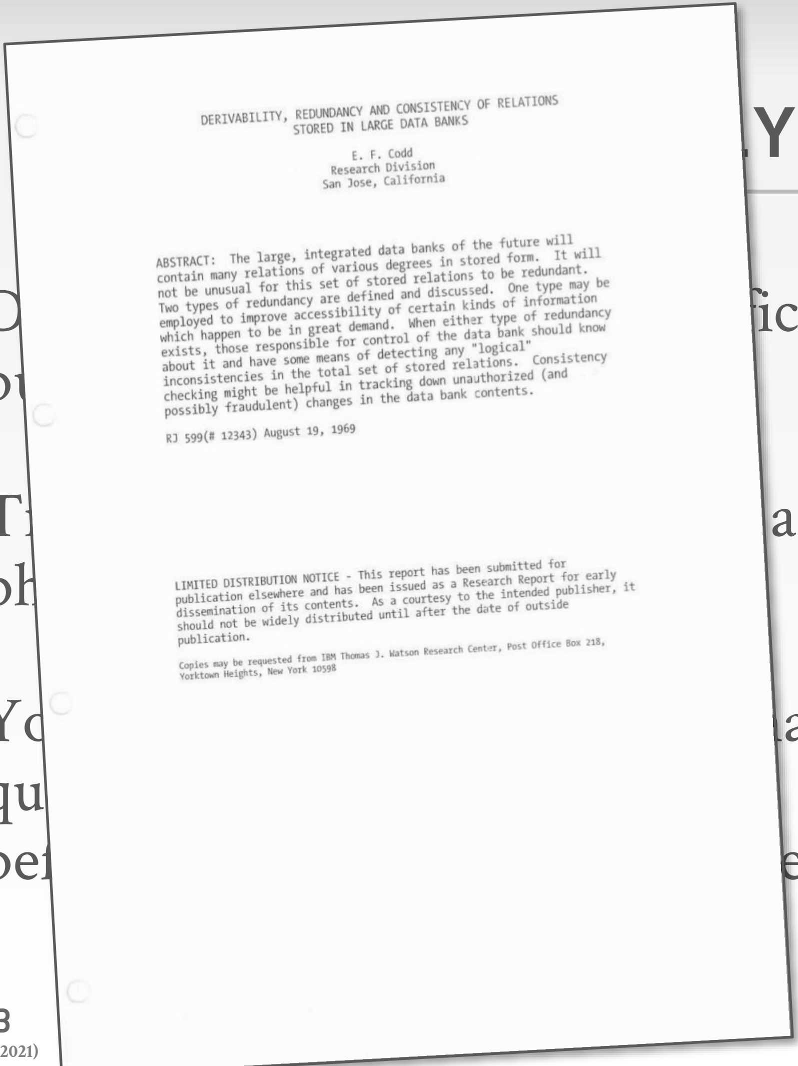
Edgar F. Codd

# Y DBMSs

D ficult to
b

T and
ph

Y at
qu
bef e.



DERIVABILITY, REDUNDANCY AND CONSISTENCY OF RELATIONS
STORED IN LARGE DATA BANKS

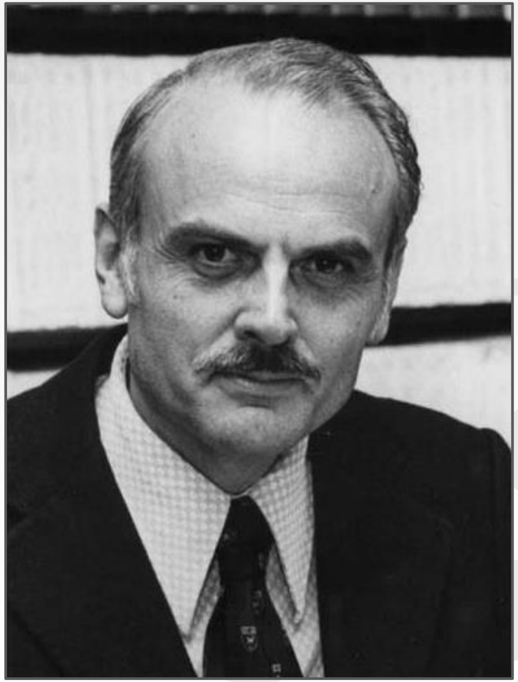
E. F. Codd
Research Division
San Jose, California

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 type may be employed to improve accessibility of certain kinds of information which happen to be in 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 total set of stored relations. Consistency checking might be helpful in tracking down unauthorized (and possibly fraudulent) changes in the data bank contents.

RJ 599(# 12343) August 19, 1969

LIMITED DISTRIBUTION NOTICE - This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from IBM Thomas J. Watson Research Center, Post Office Box 218, Yorktown Heights, New York 10598



Edgar F. Codd

Y DB

D ficult to

bu

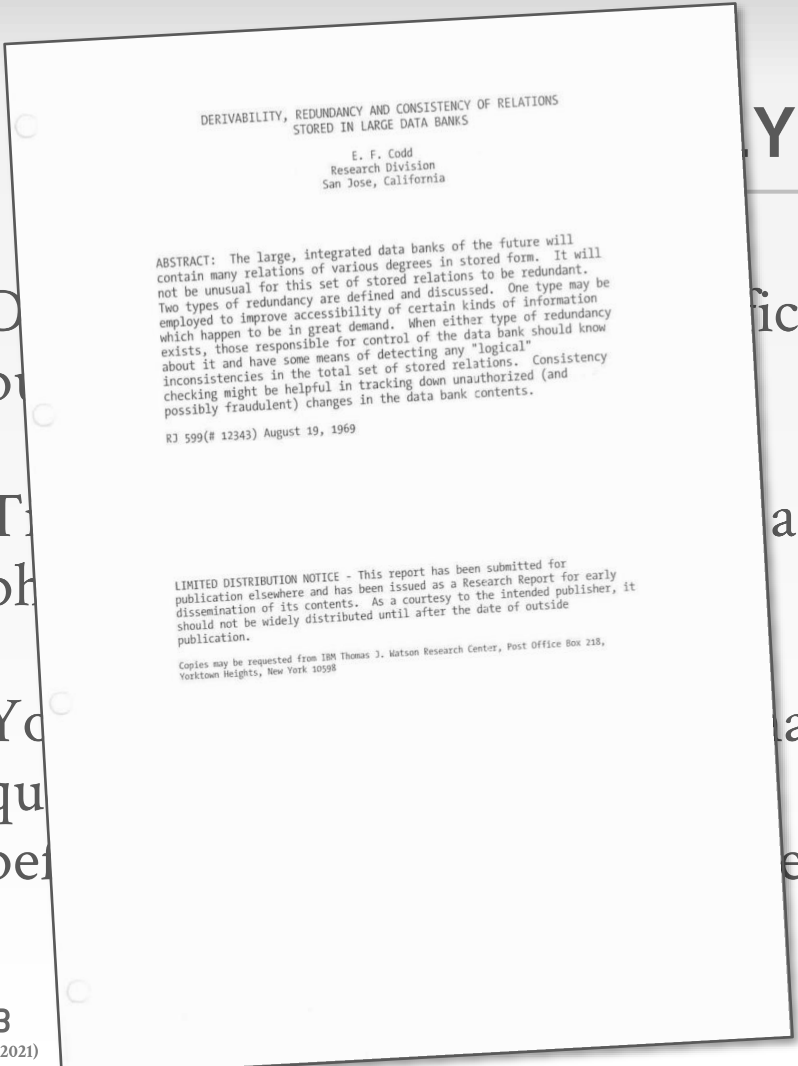
T and

ph

Yo at

qu

be e.



DERIVABILITY, REDUNDANCY AND CONSISTENCY OF RELATIONS
STORED IN LARGE DATA BANKS

E. F. Codd
Research Division
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 type may be employed to improve accessibility of certain kinds of information which happen to be in 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 total set of stored relations. Consistency checking might be helpful in tracking down unauthorized (and possibly fraudulent) changes in the data bank contents.

RJ 599(# 12343) August 19, 1969

LIMITED DISTRIBUTION NOTICE - This report has been submitted for publication elsewhere and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed until after the date of outside publication.

Copies may be requested from IBM Thomas J. Watson Research Center, Post Office Box 218, Yorktown Heights, New York 10598

*Information Retrieval* — P. BAXENDALE, Editor

**A Relational Model of Data for Large Shared Data Banks**

E. F. Codd
*IBM Research Laboratory, 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 not 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.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary 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, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity

CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

**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 Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Levein and Maron [2] 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 data representation—and certain kinds of *data inconsistency* which are expected to become troublesome even in nondeductive 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] presently in vogue for non-inferential 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 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 confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection 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 recently 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 collections of data (as opposed to individual items). Three 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 participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those 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 addresses. For example, the records of a file concerning 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 subordering of) the

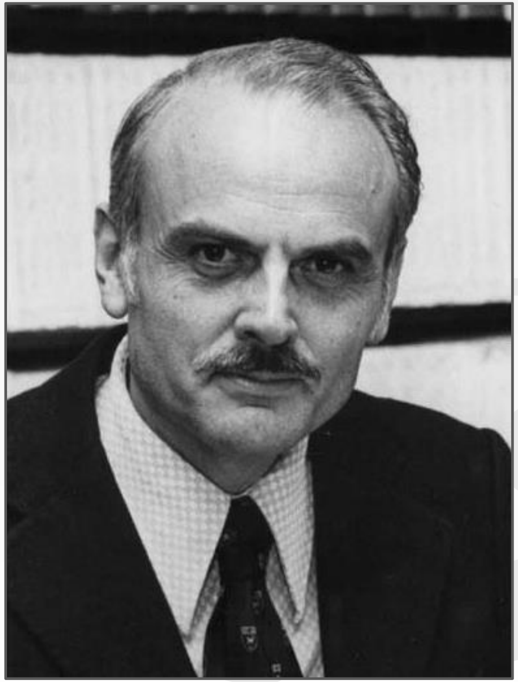
Volume 13 / Number 6 / June, 1970 — Communications of the ACM 377

# 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
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

← NoSQL

Array / Matrix

Hierarchical

Network

Multi-Value

# 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.

# 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.

`Artist(name, year, country)`

| name | year | country |
|---|---|---|
| Wu-Tang Clan | 1992 | USA |
| Notorious BIG | 1992 | USA |
| Ice Cube | 1989 | USA |

# 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.

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

| name | year | country |
|---|---|---|
| Wu-Tang Clan | 1992 | USA |
| Notorious BIG | 1992 | USA |
| Ice Cube | 1989 | USA |

*n*-ary Relation
=
Table with *n* columns

# RELATIONAL MODEL: PRIMARY KEYS

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)

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

| name | year | country |
|---|---|---|
| Wu-Tang Clan | 1992 | USA |
| Notorious BIG | 1992 | USA |
| Ice Cube | 1989 | USA |

# RELATIONAL MODEL: PRIMARY KEYS

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)

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

| id | name | year | country |
|---|---|---|---|
| 123 | Wu-Tang Clan | 1992 | USA |
| 456 | Notorious BIG | 1992 | USA |
| 789 | Ice Cube | 1989 | USA |

# RELATIONAL MODEL: FOREIGN KEYS

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)

| id | name | year | country |
|---|---|---|---|
| 123 | Wu-Tang Clan | 1992 | USA |
| 456 | Notorious BIG | 1992 | USA |
| 789 | Ice Cube | 1989 | USA |

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

| id | name | artists | year |
|---|---|---|---|
| 11 | Enter the Wu-Tang | 123 | 1993 |
| 22 | St.Ides Mix Tape | ??? | 1994 |
| 33 | AmeriKKKa's Most Wanted | 789 | 1990 |

# RELATIONAL MODEL: FOREIGN KEYS

**Artist**(<u>id</u>, name, year, country)

| id | name | year | country |
|---|---|---|---|
| 123 | Wu-Tang Clan | 1992 | USA |
| 456 | Notorious BIG | 1992 | USA |
| 789 | Ice Cube | 1989 | USA |

**Album**(<u>id</u>, name, artists, year)

| id | name | artists | year |
|---|---|---|---|
| 11 | Enter the Wu-Tang | 123 | 1993 |
| 22 | St.Ides Mix Tape | ??? | 1994 |
| 33 | AmeriKKKa's Most Wanted | 789 | 1990 |

# RELATIONAL MODEL: FOREIGN KEYS

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

| id | name | year | country |
|---|---|---|---|
| 123 | Wu-Tang Clan | 1992 | USA |
| 456 | Notorious BIG | 1992 | USA |
| 789 | Ice Cube | 1989 | USA |

**ArtistAlbum**(artist_id, album_id)

| artist_id | album_id |
|---|---|
| 123 | 11 |
| 123 | 22 |
| 789 | 22 |
| 456 | 22 |

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

| id | name | artists | year |
|---|---|---|---|
| 11 | Enter the Wu-Tang | 123 | 1993 |
| 22 | St.Ides Mix Tape | ??? | 1994 |
| 33 | AmeriKKKa's Most Wanted | 789 | 1990 |

# RELATIONAL MODEL: FOREIGN KEYS

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

| id | name | year | country |
|---|---|---|---|
| 123 | Wu-Tang Clan | 1992 | USA |
| 456 | Notorious BIG | 1992 | USA |
| 789 | Ice Cube | 1989 | USA |

**ArtistAlbum**(artist_id, album_id)

| artist_id | album_id |
|---|---|
| 123 | 11 |
| 123 | 22 |
| 789 | 22 |
| 456 | 22 |

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

| id | name | year |
|---|---|---|
| 11 | Enter the Wu-Tang | 1993 |
| 22 | St.Ides Mix Tape | 1994 |
| 33 | AmeriKKKa's Most Wanted | 1990 |

# RELATIONAL MODEL: FOREIGN KEYS

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

| id | name | year | country |
|---|---|---|---|
| 123 | Wu-Tang Clan | 1992 | USA |
| 456 | Notorious BIG | 1992 | USA |
| 789 | Ice Cube | 1989 | USA |

**ArtistAlbum**(artist_id, album_id)

| artist_id | album_id |
|---|---|
| 123 | 11 |
| 123 | 22 |
| 789 | 22 |
| 456 | 22 |

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

| id | name | year |
|---|---|---|
| 11 | Enter the Wu-Tang | 1993 |
| 22 | St.Ides Mix Tape | 1994 |
| 33 | AmeriKKKa's Most Wanted | 1990 |

# 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.

# 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.

← Relational Algebra

**Non-Procedural (Declarative):**

→ The query specifies only what data is wanted and not how to find it.

# 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.

← Relational Algebra

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

← Relational Calculus

# 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.

← Relational Algebra

**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.

| | |
|---|---|
| $\sigma$ | Select |
| $\pi$ | Projection |
| $\cup$ | Union |
| $\cap$ | Intersection |
| $-$ | Difference |
| $\times$ | Product |
| $\bowtie$ | Join |

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

# RELATIONAL ALGEBRA: SELECT

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a2 | 103 |
| a3 | 104 |

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

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a2 | 103 |
| a3 | 104 |

$\sigma_{a\_id='a2'}(R)$

| a_id | b_id |
|---|---|
| a2 | 102 |
| a2 | 103 |

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

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a2 | 103 |
| a3 | 104 |

$\sigma_{a\_id='a2'}(R)$

| a_id | b_id |
|---|---|
| a2 | 102 |
| a2 | 103 |

$\sigma_{a\_id='a2' \land b\_id>102}(R)$

| a_id | b_id |
|---|---|
| a2 | 103 |

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

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a2 | 103 |
| a3 | 104 |

$\sigma_{a\_id='a2'}(R)$

| a_id | b_id |
|---|---|
| a2 | 102 |
| a2 | 103 |

$\sigma_{a\_id='a2' \land b\_id>102}(R)$

| a_id | b_id |
|---|---|
| a2 | 103 |

```
SELECT * FROM R
 WHERE a_id='a2' AND b_id>102;
```

# RELATIONAL ALGEBRA: PROJECTION

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a2 | 103 |
| a3 | 104 |

Generate a relation with tuples that contains only the specified attributes.

→ Can rearrange attributes' ordering.

→ Can manipulate the values.

**Syntax:** $\pi_{A1,A2,\ldots,An}(R)$

# RELATIONAL ALGEBRA: PROJECTION

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

**Syntax:** $\pi_{A1,A2,...,An}(R)$

R(a_id,b_id)

| a_id | b_id |
| --- | --- |
| a1 | 101 |
| a2 | 102 |
| a2 | 103 |
| a3 | 104 |

$\pi_{b\_id-100,\,a\_id}(\sigma_{a\_id='a2'}(R))$

| b_id-100 | a_id |
| --- | --- |
| 2 | a2 |
| 3 | a2 |

# RELATIONAL ALGEBRA: PROJECTION

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

**Syntax:** $\pi_{A1,A2,\ldots,An}(R)$

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a2 | 103 |
| a3 | 104 |

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

| b_id-100 | a_id |
|---|---|
| 2 | a2 |
| 3 | a2 |

```
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 U S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

# RELATIONAL ALGEBRA: UNION

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

**Syntax:** **(R ∪ S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R ∪ S)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

# RELATIONAL ALGEBRA: UNION

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

**Syntax:** (R ∪ S)

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R ∪ S)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

```
(SELECT * FROM R)
UNION ALL
(SELECT * FROM S);
```

# RELATIONAL ALGEBRA: INTERSECTION

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

**Syntax: (R ∩ S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

# RELATIONAL ALGEBRA: INTERSECTION

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

**Syntax: (R ∩ S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R ∩ S)

| a_id | b_id |
|---|---|
| a3 | 103 |

```
(SELECT * FROM R)
INTERSECT
(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)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

# 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)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R - S)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |

# 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)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R - S)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |

```
(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 × S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

# RELATIONAL ALGEBRA: PRODUCT

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

**Syntax:** **(R × S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R × S)

| R.a_id | R.b_id | S.a_id | S.b_id |
|---|---|---|---|
| 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 |

# RELATIONAL ALGEBRA: PRODUCT

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

**Syntax: (R × S)**

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

```
SELECT * FROM R, S;
```

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R × S)

| R.a_id | R.b_id | S.a_id | S.b_id |
|---|---|---|---|
| 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 |

# 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 ⋈ S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

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

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

$(R \bowtie S)$

| a_id | b_id |
|---|---|
| a3 | 103 |

# 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 ⋈ S)**

R(a_id,b_id)

| a_id | b_id |
|---|---|
| a1 | 101 |
| a2 | 102 |
| a3 | 103 |

S(a_id,b_id)

| a_id | b_id |
|---|---|
| a3 | 103 |
| a4 | 104 |
| a5 | 105 |

(R ⋈ S)

| a_id | b_id |
|---|---|
| a3 | 103 |

```
SELECT * FROM R NATURAL JOIN S;
```

# RELATIONAL ALGEBRA: EXTRA OPERATORS

Rename (ρ)

Assignment (R←S)

Duplicate Elimination (δ)

Aggregation (γ)

Sorting (τ)

Division (R÷S)

# OBSERVATION

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

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

# OBSERVATION

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.

# RELATIONAL MODEL: QUERIES

The relational model is independent of any query language implementation.

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

# RELATIONAL MODEL: QUERIES

The relational model is independent of any query language implementation.

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

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

# RELATIONAL MODEL: QUERIES

The relational model is independent of any query language implementation.

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

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

```
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