

## Lecture 3.

# **Relational data model and relational DBMSs**

# Computers in the Early 1970s



- Computers were being used across a wide range of industries.
- **Minicomputers** (e.g., DEC's PDP series) became affordable for small businesses.
- Systems became **interactive**, moving beyond batch processing.
- **Multi-user operating systems** and **file systems** saw significant development.
- **Network and hierarchical DBMSs** were widely adopted.
- A diverse ecosystem of programming languages emerged.
- Early concepts appeared about enabling non-professionals to use computers.

# **Navigational Approach - Pros and Cons**

# Navigational Approach in Early DBMSs

## Strengths of the Navigational Approach:

- **High Performance for Known Access Paths.** Direct pointer-based navigation enabled extremely fast traversal along predefined relationships.
- **Efficient Use of Limited Hardware Resources.** Optimized for mainframes with minimal memory and storage—no query parsing or optimization overhead.
- **Fine-Grained Control for Developers.** Programmers could precisely manage data access and transaction flow, crucial for mission-critical systems.
- **Predictable Behavior.** Execution was deterministic and transparent—ideal for real-time and embedded applications (e.g., IMS in NASA's Apollo program).

# Navigational Approach in Early DBMSs

## Weaknesses of the Navigational Approach:

- **Complex and Error-Prone Programming.** Developers had to manually code every step of data traversal, increasing development time and bug risk.
- **Inflexible Schema Changes.** Modifying data structures often required rewriting large portions of application logic.
- **Poor Support for Ad Hoc Queries.** No declarative query language (like SQL)—new queries demanded new code.
- **Tight Coupling Between Code and Data Structure.** Applications were highly dependent on physical storage layout, reducing maintainability and portability.

# Problems of the Navigational Approach

- **DBMS as a programmer's tool:**

Requires highly skilled developers to use effectively.

- **Code required for every query:**

Any data request must be implemented through custom programming logic—no ad hoc querying.

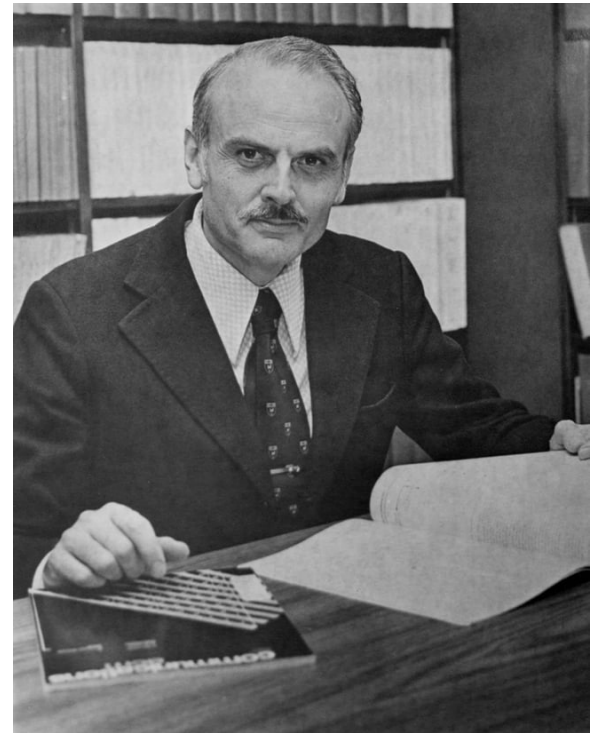
- **Tight coupling of logical and physical data layers:**

Logical data structure is intertwined with its physical storage implementation, reducing flexibility and maintainability.

# **Edgar F. Codd's Critique of the Navigational Approach**

## Edgar Frank "Ted" Codd (1923–2003)

- Held a PhD in Computer Science and Computational Engineering
- Joined IBM's Research Center in 1965
- Proposed a revolutionary new approach to database organization
- Awarded the Turing Award in 1981 for his foundational work on the relational model



# Recognizing the Problem

Edgar F. Codd identified critical flaws in the navigational approach:

- **Complexity & Inaccessibility.** Only highly skilled programmers could write and maintain data access code—limiting DBMS use to specialists.
- **Lack of Data Independence.** Applications were tightly coupled to physical storage structures; any change required code rewrites.
- **No Declarative Querying.** Users had to specify how to retrieve data (procedural navigation), not just what they needed.
- **Poor Flexibility for Ad Hoc Queries.** New questions required new programs—making exploratory or dynamic data analysis impractical.

Codd realized databases should be based on solid mathematical foundations, separating logic from implementation—and accessible to non-programmers.

# Codd's Relational Model

# The Relational Model: Origins

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

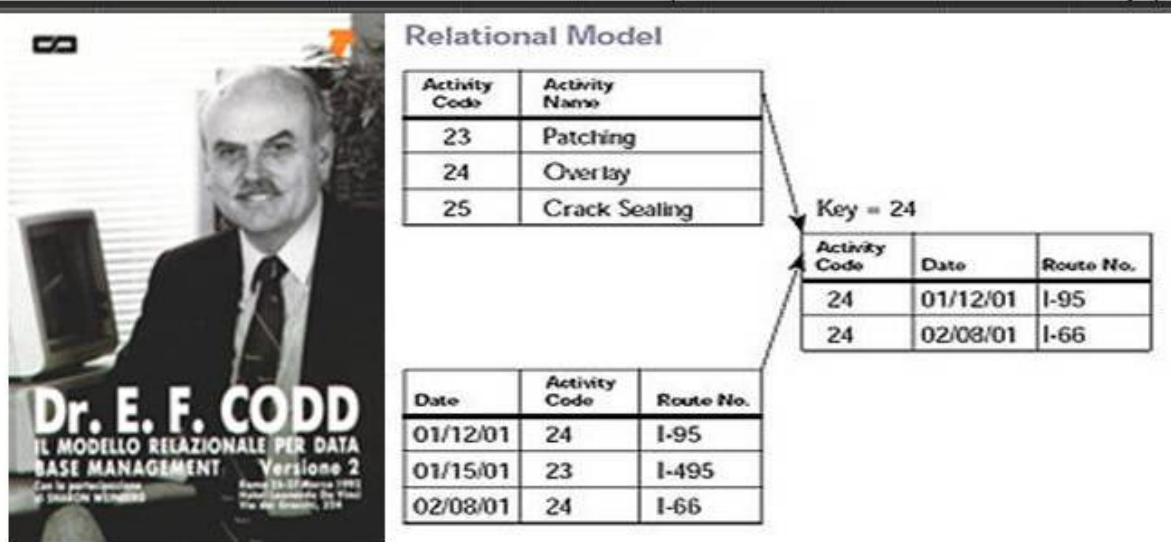
E. F. Codd

*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

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”).



«A Relational Model of Data for Large Shared Data Banks»  
Computer Machinery journal (1970 год)

Association of

# Key Theses of the Paper

## A Radical New Vision - **Data as Relations, Not Pointers**

- Proposed representing all **data as mathematical relations (tables)**, not as linked records or hierarchical trees.
- Rejected physical pointers and navigational paths—data access should be based on values, not addresses.
- Aimed to shield users from the internal storage structure (“protect users from having to know how the data is organized in the machine”).

# Key Theses of the Paper

## The Power of Simplicity - **One Universal Structure**

- All data—entities and relationships—expressed through **n-ary relations** (tables with rows and columns).
- Eliminated need for separate constructs like “sets” (network) or “parent-child” links (hierarchical).
- Provided a uniform, simple model applicable to any domain.

# Key Theses of the Paper

## Data Independence - **Logical** ≠ **Physical**

- Introduced clear separation between **logical data model** (what users see) and **physical storage** (how data is stored).
- Changes to storage (e.g., adding indexes) should **not affect application code**.
- Enabled true data **independence**—a foundational principle of modern databases

# Key Theses of the Paper

## Declarative Access - “What,” Not “How”

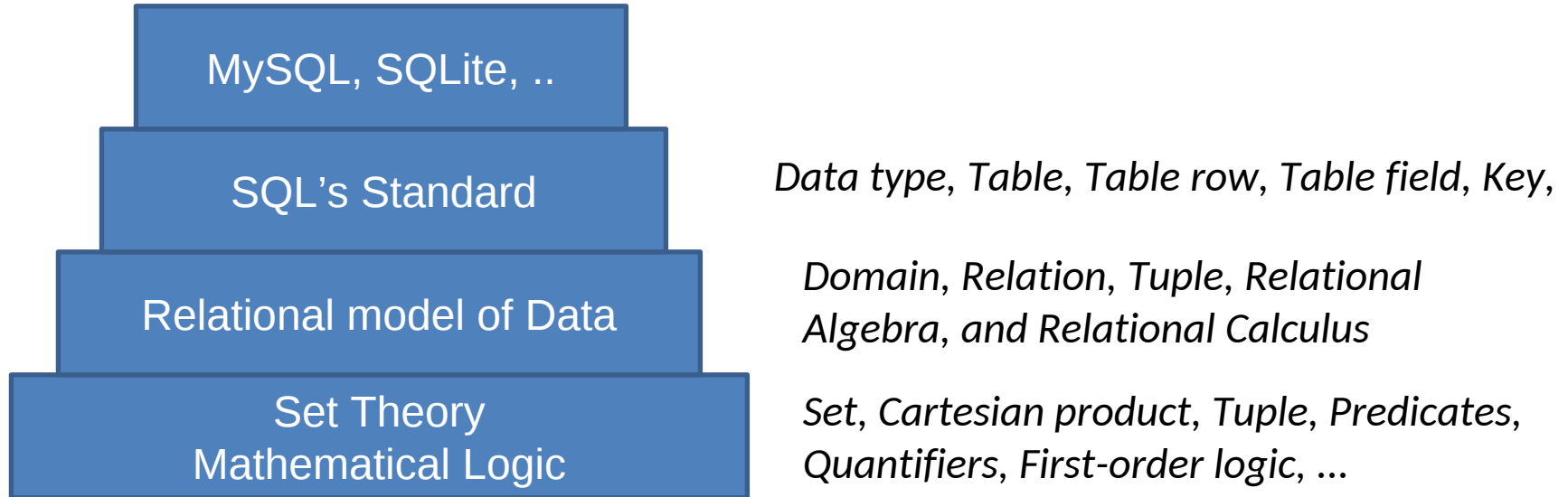
- Advocated for a **universal data sublanguage** based on predicate logic (later realized as SQL).
- Users specify **what data they want**, not **how to retrieve it** (no procedural navigation).
- Made databases accessible to **non-programmers** and enabled ad hoc querying.

# Key Theses of the Paper

## Foundation for Integrity & Flexibility - **Normalization and Beyond**

- Introduced the concept of **normal forms** to reduce redundancy and prevent update anomalies.
- Emphasized **data integrity** through relational constraints (though formalized later).
- Laid the groundwork for a **flexible, mathematically sound** foundation for large, shared data banks

# Mathematical Foundations of the Relational Model



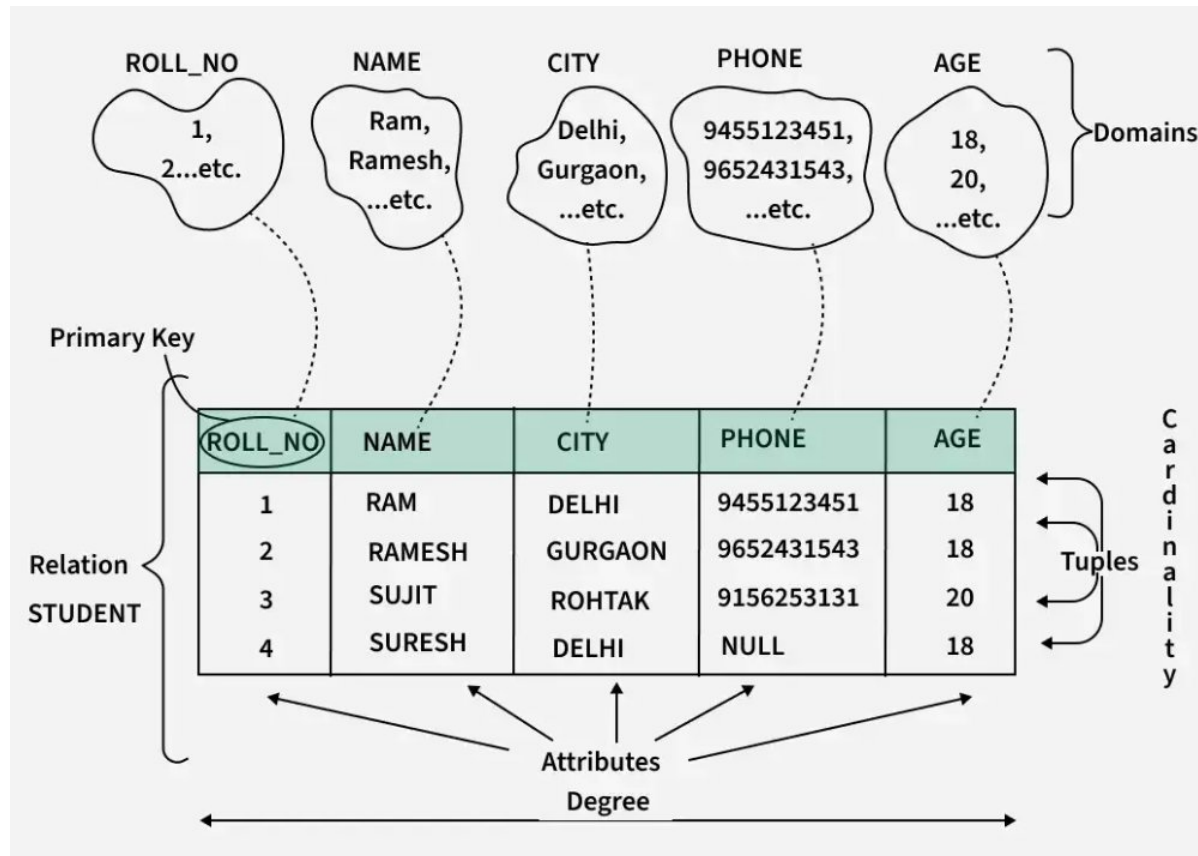
The relational data model is based on a rigorous mathematical theory

# **Core Concepts and Terms of the Relational Model**

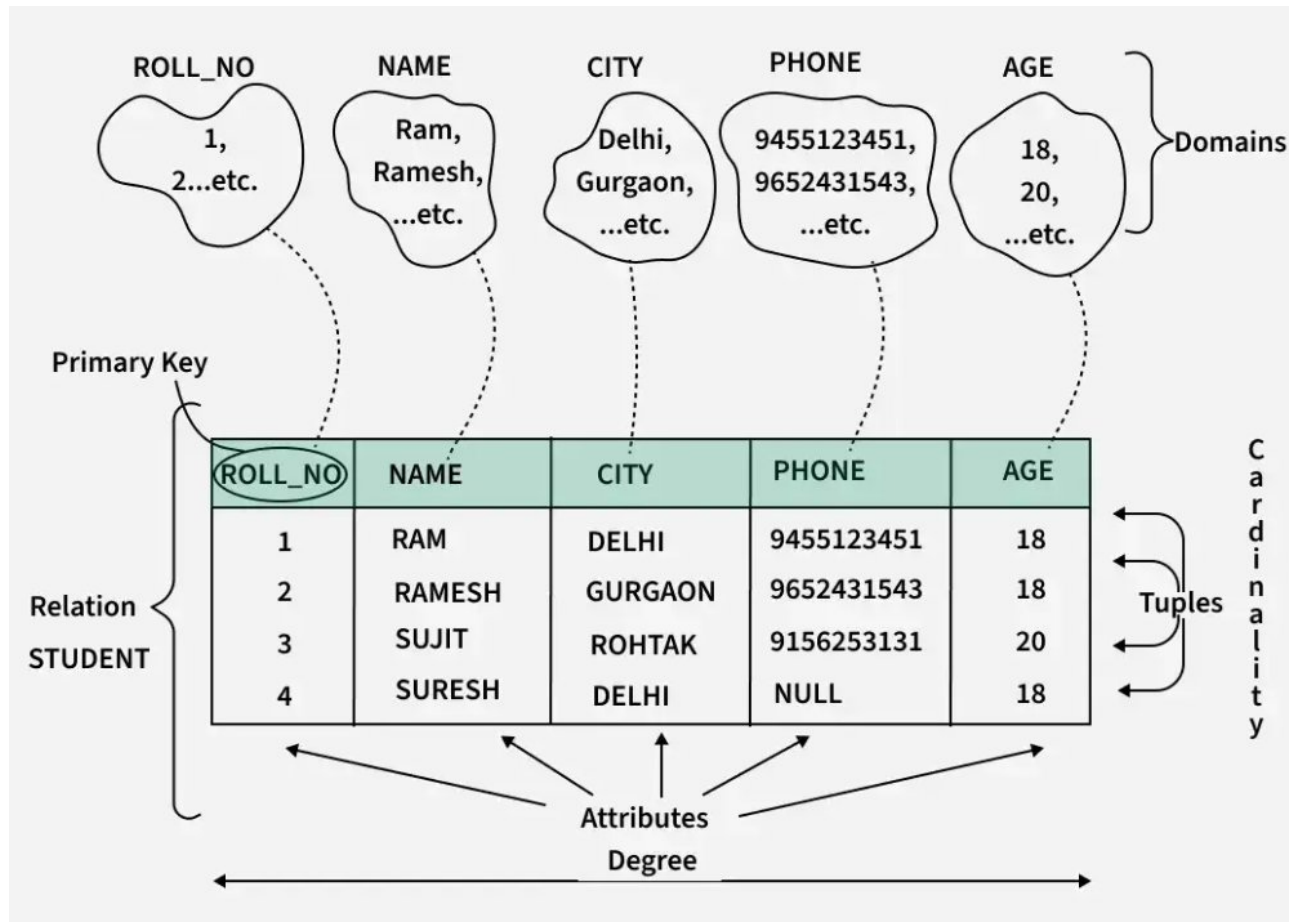
# Relational Model

The relational model represents how data is stored and managed in Relational Databases where data is organized into **tables**, each known as a **relation**.

Each **row** of a table represents an **entity** or **record** and each **column** represents a particular **attribute** of that entity.



# Relation

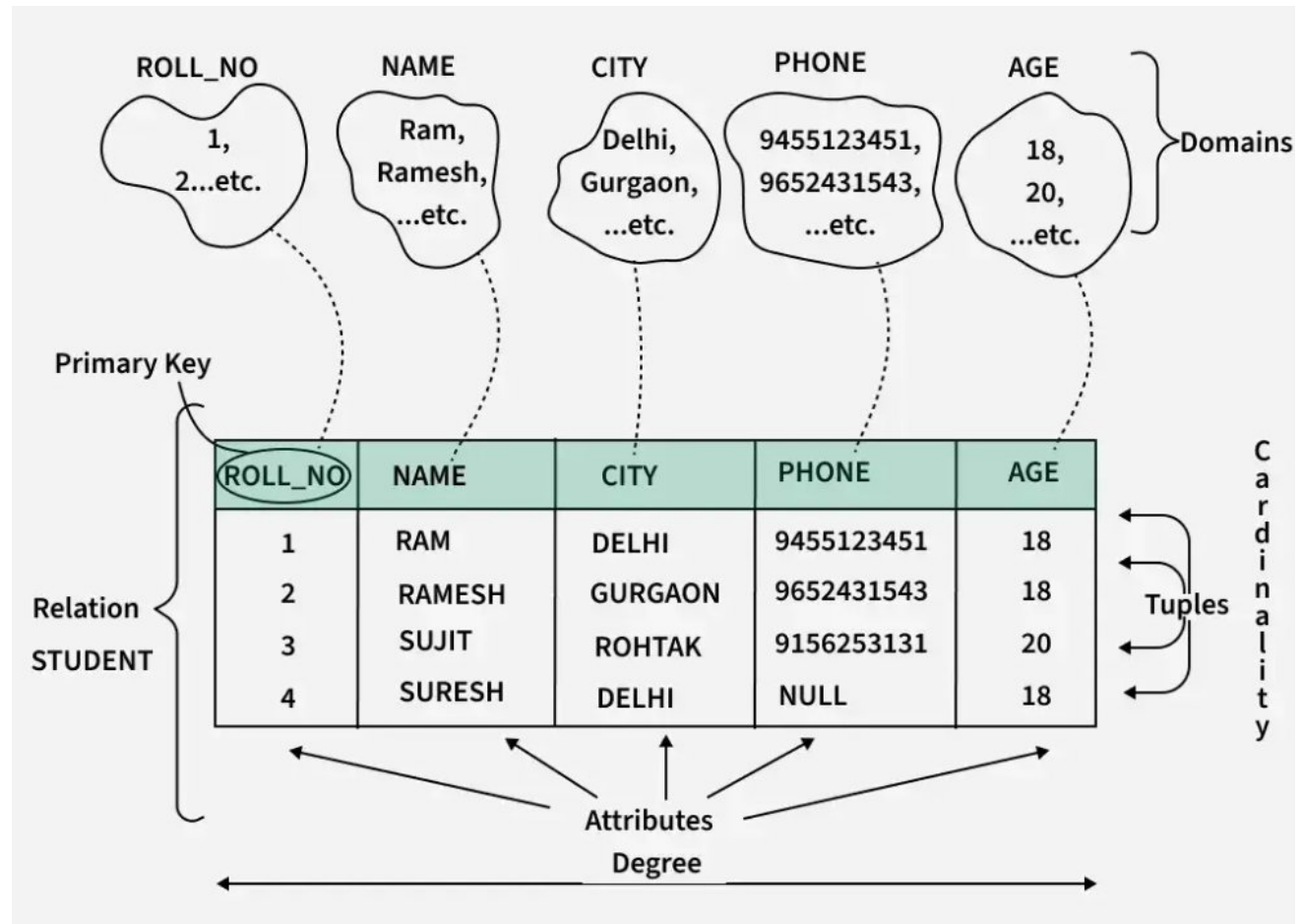


A **relation** is just a **table** (based on Math!)

- In mathematics, a relation is a set of related facts.
- In databases, a relation = a table with a fixed structure
  - No duplicate rows (a set can't contain identical elements).
  - Row order doesn't matter—only the data itself counts.

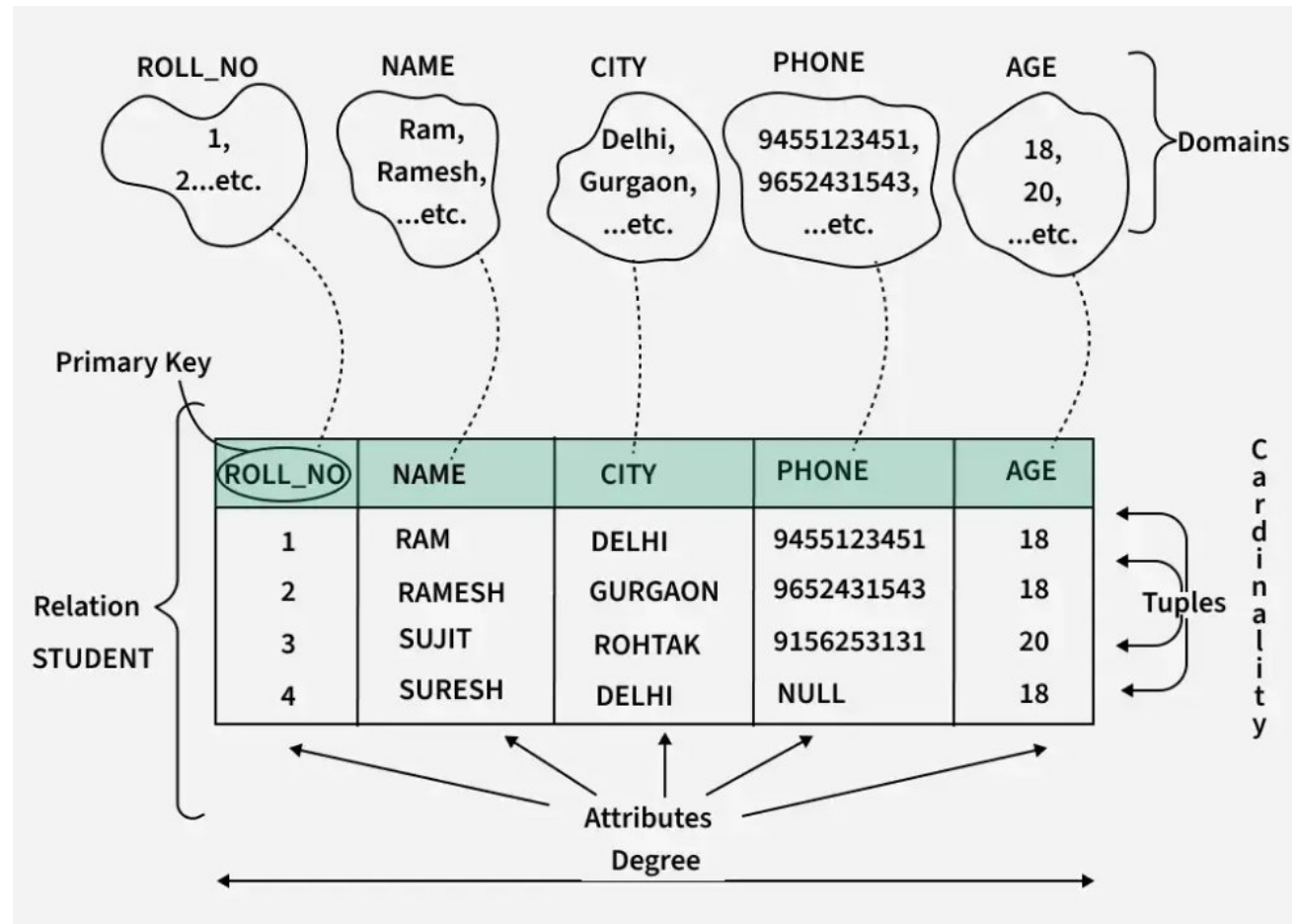
# Attributes and Domains

---



- An **attribute** is a named column in a relation.
- Each attribute has a **domain**—the set of allowed values (e.g., integers, text, dates).
- Attributes define **what kind of data** the relation stores.

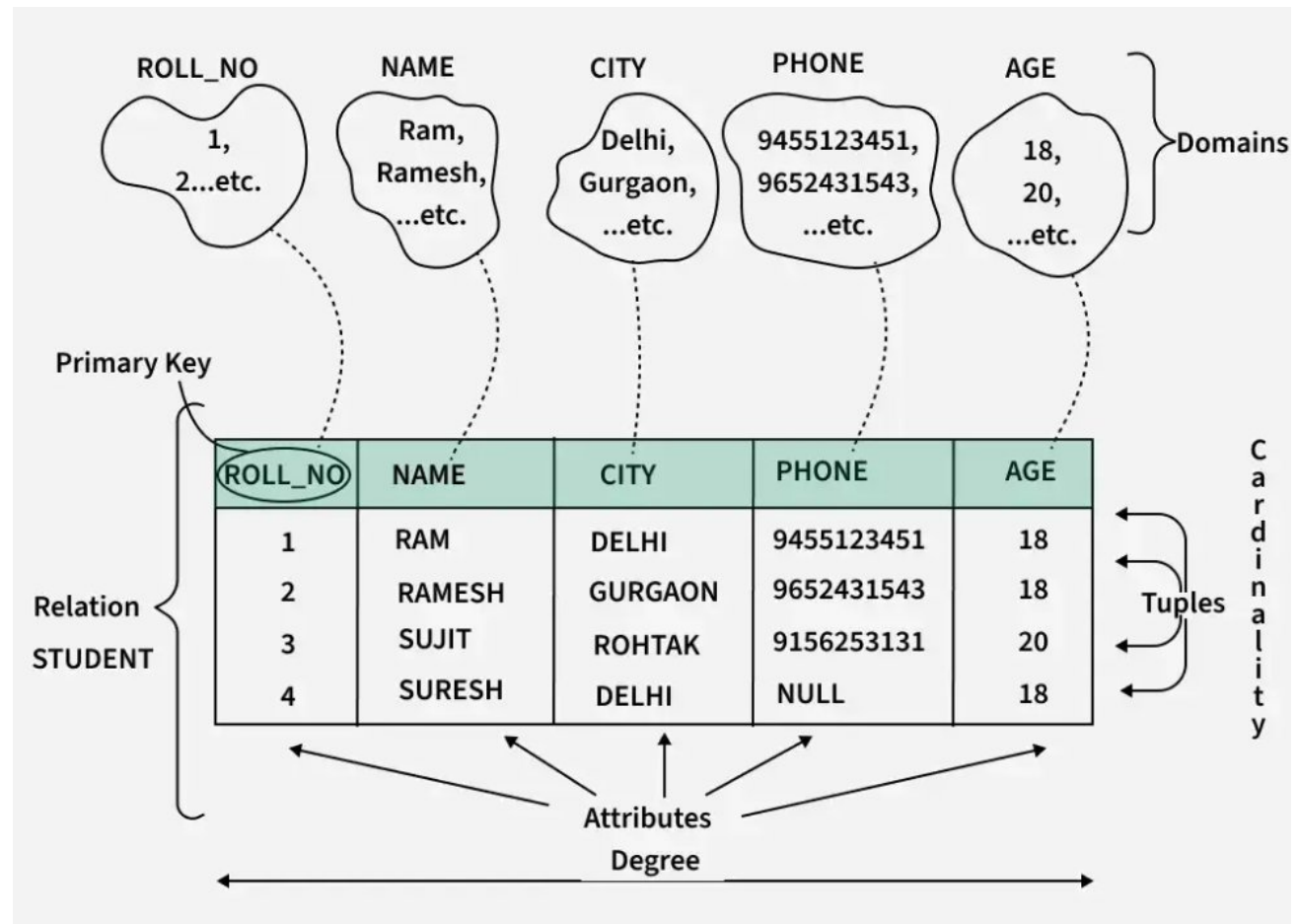
# Relation Schema



A **relation schema** defines the structure of the relation and represents the **name of the relation with its attributes**.

STUDENT (ROLL\_NO, NAME, ADDRESS, PHONE and AGE) is the relation schema for STUDENT.

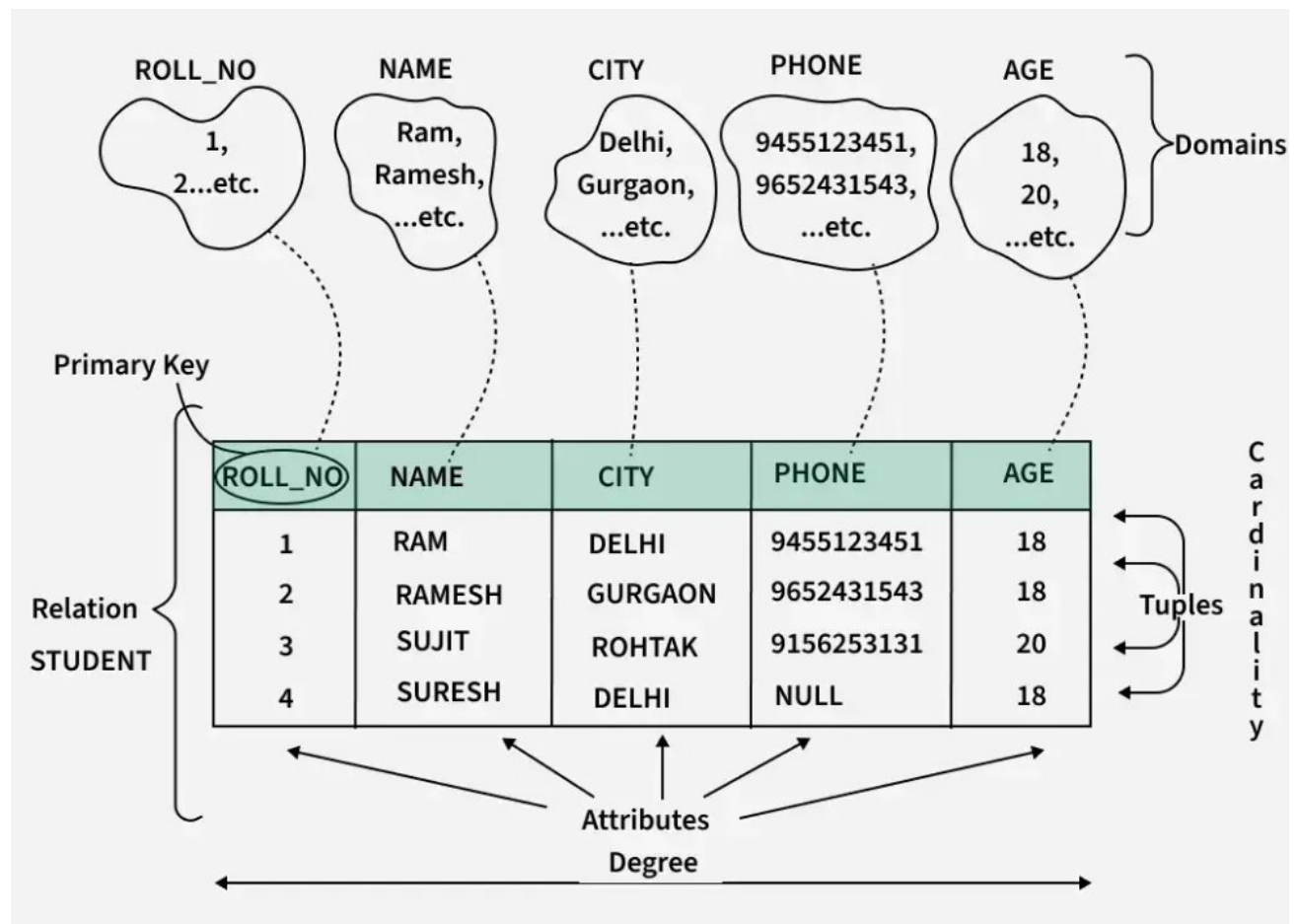
# Tuple



- A **tuple** is a single entry in a relation—one full record.
- It contains one value for each attribute (column).
- Represents one real-world entity or event (e.g., one employee, one product).

(1, RAM, DELHI, 9455123451, 18) is a tuple in the STUDENT table.

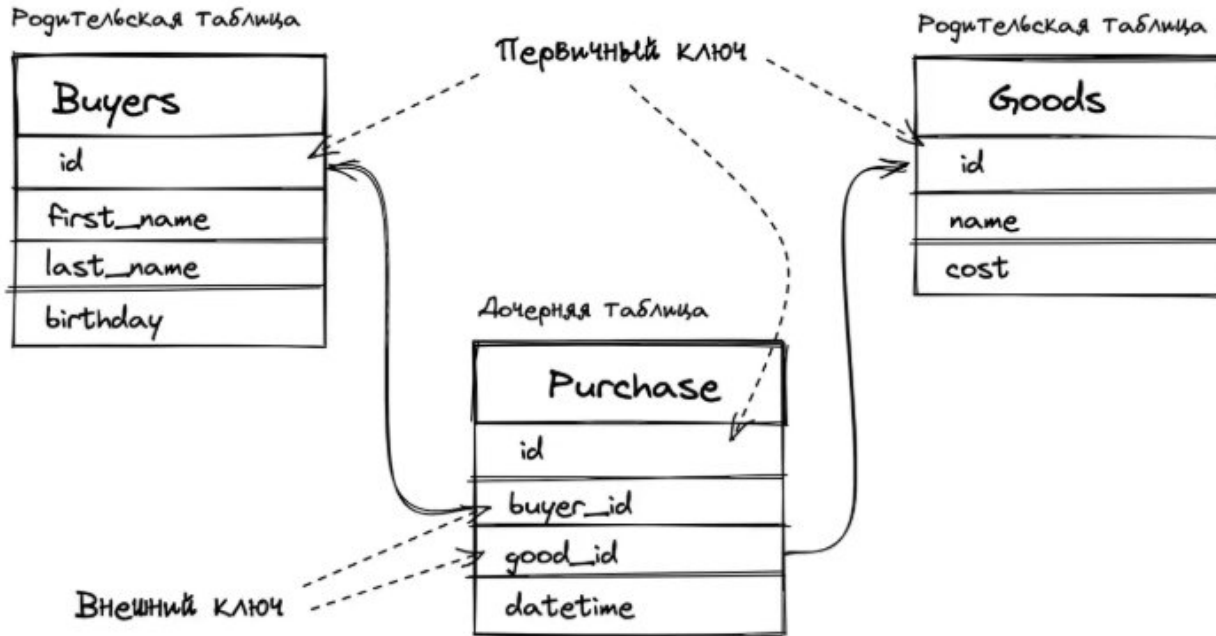
# Primary key



- A **primary key** is one (or more) attribute(s) that **uniquely identify** every tuple in a relation.
- No two rows can have the same primary key value.
- Guarantees **each record is distinct and findable**.

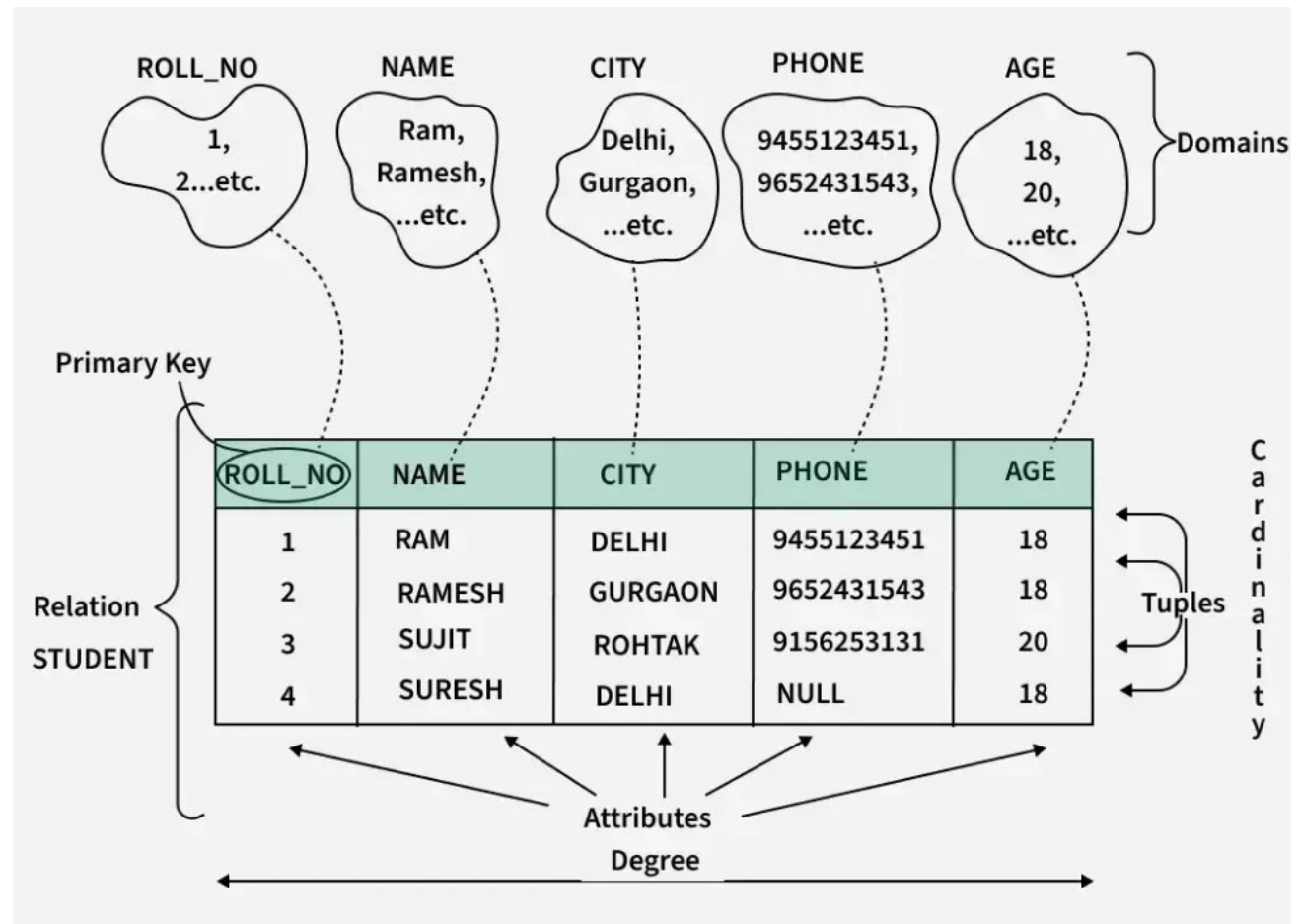
ROLL\_NO is the primary key in the STUDENT table.

# Different Kinds of Keys



- **Superkey** – a set of attributes that uniquely identifies a tuple.
- **Candidate key** – a minimal superkey (i.e., no unnecessary attributes).
- **Primary key** – the candidate key chosen to uniquely identify tuples in a relation.
- **Foreign key** – an attribute (or set of attributes) in one table that refers to the primary key of another table (or the same table).

# Degree and Cardinality



- **Degree:** The number of attributes in the relation is known as the degree of the relation.
- **Cardinality:** The number of tuples in a relation is known as cardinality.

The STUDENT relation has a degree of 5 and cardinality 4.

# Relational Algebra

# What Is Relational Algebra?

Relational algebra is a formal system for querying tables.

- Relational algebra is a **set of mathematical operations** that work on **relations (tables)**.
- Each operation takes one or more tables as input and produces a **new table** as output.
- It forms the **theoretical foundation** for declarative query languages like SQL.

Think of it as “arithmetic for tables”—you combine and transform data using precise rules.

# Why It Matters

Relational algebra is the engine behind declarative queries.

- Users write **what they want** (e.g., “Show all customers in London”).
- The DBMS uses relational algebra to **plan how to get it**.
- Enables **optimization**: the system can choose the fastest sequence of operations.

Relational algebra makes high-level, user-friendly querying possible—without writing step-by-step code.

# Core Operations – Selection & Projection

---

- **Selection ( $\sigma$ )** – *Filter rows*

Example:  `$\sigma_{city='London'}(Customers)$`  → returns only customers from London.

- **Projection ( $\pi$ )** – *Choose columns*

Example:  `$\pi_{name, email}(Customers)$`  → returns just name and email for all customers.

📌 *Selection = “keep these rows”; Projection = “keep these columns.”*

## Core Operations – Union, Difference, Intersection

- **Union (u)** – Combine rows from two compatible tables (no duplicates).
- **Difference (-)** – Find rows in Table A that aren't in Table B.
- **Rename (p)** – Change a table's or column's name for clarity or reuse.

 *Tables must have the same number and types of columns to use union or difference.*

# Core Operations – Union, Difference, Intersection

**Table A: Active Customers**

id	name	↓
1	Alice	
2	Bob	

**Table B: VIP Customers**

id	name	↓
2	Bob	
3	Carol	

# Core Operations – Union, Difference, Intersection

---

- **Union ( $A \cup B$ )** – All customers who are *active* OR *VIP*:

id	name	⌵
1	Alice	
2	Bob	
3	Carol	

- **Difference ( $A - B$ )** – Active but *not* *VIP*:

id	name	⌵
1	Alice	

- **Intersection ( $A \cap B$ )** – Both active *and* *VIP*:

id	name	⌵
2	Bob	

# Join – The Power Operation

---

- **Join** (⋈) – Combine related data from two tables using a common attribute.

## Orders

order_id	cust_id	⬇
101	1	200
102	2	150

## Customers

id	name	⬇
1	Alice	
2	Bob	

# Join – The Power Operation

---

Result of `Orders ⋈_{cust_id=id} Customers`:

order_id	cust_id	amount	name
101	1	200	Alice
102	2	150	Bob

 *Join links tables by matching values—no physical pointers required!*

 *This is how relational databases connect information logically.*

# **Relational Data Model - Definition and Significance**

The relational data model is a mathematical model based on set theory and first-order logic, proposed by Edgar Codd in 1970.

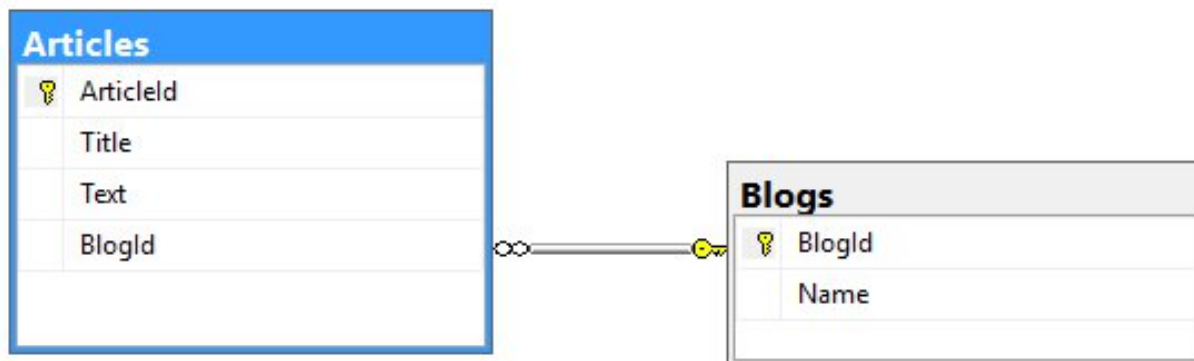
- It represents data as **relations (tables)** composed of **tuples (rows)**, where each **attribute (column)** belongs to a specific **domain (a set of allowed values)**.
- Data manipulation operations are formally defined through **relational algebra** and **relational calculus**.
- Relationships between tables are established **on-the-fly** during query execution—specifically through **JOIN operations** that match values (typically via foreign keys), rather than through physical pointers or pre-defined links.

# What does “linked on-the-fly” mean?


---

Relational algebra is the engine behind declarative queries.

- There are no physical links between rows in different tables.
- There are no predefined access paths (unlike in hierarchical or network models).
- Relationships are established dynamically at query time by matching values (e.g., foreign keys).
- You can join any tables as long as there are semantically compatible columns (e.g., matching IDs).
- You can perform arbitrary combinations, aggregations, and filters—offering flexibility at the logical level, not the physical one.



The relational model sparked a revolution: it made data independent of physical storage, declarative in nature, and mathematically rigorous.

Aspect	Hierarchical / Network Models	Relational Model 
<b>Type of Relationships</b>	Fixed, predefined pointers (parent-child or owner-member sets). Physical links embedded in storage.	Logical relationships via matching values (e.g., foreign keys). No physical links—relationships are established dynamically.
<b>Query Flexibility</b>	Low: queries must follow predefined paths; ad hoc queries require custom code.	High: declarative queries (e.g., SQL) allow arbitrary joins, filters, and aggregations without preplanning.
<b>Data Independence</b>	Low: applications tightly coupled to physical storage structure.	High: logical schema is separate from physical storage; changes to storage don't affect applications.
<b>Schema Modification</b>	Difficult: adding new relationships or entity types often requires rewriting programs and restructuring data.	Easier: schema can evolve with minimal impact on existing queries (e.g., adding columns or tables).
<b>Basis for Optimization</b>	Limited: access paths are fixed; little room for query optimization.	Strong: based on relational algebra/calculus, enabling powerful query optimization (e.g., join reordering, index selection).

# Navigational vs. Relational Approaches

The key advantage of the relational model is its use of **non-procedural (associative) data processing**.

- In the relational model, database queries are not constrained by physical pointers, so there's no need to write custom programs to retrieve data.
- Queries continue to work correctly even after logical reorganization of the database schema.
- Relational applications are significantly simpler than navigational ones.

# **The Relational Model: From Theory to Practice**

# The Codd vs. Bachman Debate – ACM SIGMOD, 1973

---

## Edgar Codd (The Idealist)

- Championed a **mathematically pure, declarative model**.
- Believed **abstraction** → **simplicity** → **reliability** → **scalability**.
- Argued that hiding physical details empowers users and ensures long-term maintainability.

## Charles Bachman (The Pragmatist)

- Designed the **navigational IDS/DBTG network model**.
- Emphasized **performance, fine-grained control, and hardware efficiency**.
- Warned that abstraction could obscure critical system behavior.

## Core Question:

*What matters more—**simplicity through abstraction** or **power through control**?*

# The Codd vs. Bachman Debate - ACM SIGMOD, 1973

---

## **Codd's vision prevailed.**

Relational databases became the global standard—not because they were always faster, but because developer productivity, data independence, and adaptability proved more valuable in the long run.

Modern systems (even high-performance ones) now blend both ideas—but they speak Codd's language first.

# Early relational DBMS

# System R Project

Initially, IBM ignored Codd's ideas because it already had a successful, profitable, and widely deployed system—**IMS**, a hierarchical DBMS. The relational model was seen as too slow, impractical to implement, and a threat to IBM's existing business.

However, in 1973, IBM's research lab in San Jose launched the **System R project**—a prototype relational DBMS. Key innovations of System R:

- **SQL** (developed by Chamberlin and Boyce) as a user-friendly query language. Declarative, close to natural language, simple syntax.
- **Query Optimizer:** Automatically translated high-level SQL queries into efficient execution plans.
- **Query Compiler:** Cached and reused optimized query plans for better performance.

# Oracle – The First Commercial RDBMS

## Founded on IBM's Public Research

- **1977:** Larry Ellison, Bob Miner, and Ed Oates found **Software Development Laboratories** (later **Oracle Corporation**).
- They read IBM's published papers on **System R** and built their own SQL-compatible RDBMS—**Oracle V1** (released in 1979).
- **Key fact:** Oracle beat IBM to market—IBM's commercial product (SQL/DS) arrived only in **1981**.
- Oracle became the first widely adopted **commercial relational database**.

# Ingres – The Academic Challenger

---

Developed at UC Berkeley (1974–1985)

- Led by **Michael Stonebraker** and **Eugene Wong**.
- Used its own query language, **QUEL** (considered more elegant than early SQL).
- Focused on **portability, modular architecture, and interactive querying**.
- Later commercialized as **Ingres Corporation** (1980s).
- Influenced many future systems, including **Postgres, Sybase, and Microsoft SQL Server**.

# Postgres – The Object-Relational Evolution

## From Ingres to PostgreSQL

- Also created by **Michael Stonebraker** at UC Berkeley (mid-1980s).
- Goal: extend relational model with **complex data types, integrity constraints, and extensibility** → birth of the **object-relational** model.
- Name = **Post-Ingres** (“after Ingres”).
- Open-sourced in **1996** as **PostgreSQL**.
- Today: one of the most advanced, standards-compliant, open-source RDBMSs—supporting JSON, geospatial data, full-text search, and more.

These pioneering systems

- **System R**
- **Oracle**
- **Ingres**
- **Postgres**

laid the foundation for modern data management and proved that Codd's relational vision was not only sound but transformative.

## Relational Approach: Additional Advantages

Beyond improved developer productivity and ease of use:

- The relational model fits naturally into **client-server architectures**, enabling efficient exchange of high-level queries and result sets.
- The **declarative nature of SQL** allows query compilers to automatically optimize and parallelize data processing.
- Relational data maps intuitively to **graphical user interfaces**, especially spreadsheet-like views.