CSL303/MAL505 — Database Management Systems

Module I: Relational Foundations & SQL
Lecture 2: The Relational Model

Amit Kumar Dhar

July 29, 2026

IIT Bhilai

Recap of Lecture 1

- A DBMS provides a data model, a query language, integrity, transactions, concurrency, recovery, and security.

Recap of Lecture 1

- A DBMS provides a data model, a query language, integrity, transactions, concurrency, recovery, and security.
- Three-schema architecture $\rightarrow$ **data independence**.

Recap of Lecture 1

- A DBMS provides a data model, a query language, integrity, transactions, concurrency, recovery, and security.
- Three-schema architecture $\rightarrow$ **data independence**.
- Today we make the **logical level** concrete with the **relational model** — the foundation under SQL and everything else this term.

Agenda

- Why the relational model won

Agenda

- Why the relational model won
- Relations, tuples, attributes, domains

Agenda

- Why the relational model won
- Relations, tuples, attributes, domains
- Schema vs instance (precisely)

Agenda

- Why the relational model won
- Relations, tuples, attributes, domains
- Schema vs instance (precisely)
- Keys: super, candidate, primary, foreign

Agenda

- Why the relational model won
- Relations, tuples, attributes, domains
- Schema vs instance (precisely)
- Keys: super, candidate, primary, foreign
- Integrity constraints: domain, entity, referential

Introduction

A little history

- **1970** — E. F. Codd, *"A Relational Model of Data for Large Shared Data Banks."*

A little history

- **1970** — E. F. Codd, *"A Relational Model of Data for Large Shared Data Banks."*
- Radical idea: store data as simple **tables**, backed by **mathematical relations**.

A little history

- **1970** — E. F. Codd, *"A Relational Model of Data for Large Shared Data Banks."*
- Radical idea: store data as simple **tables**, backed by **mathematical relations**.
- Hide storage details; query with a **declarative** language.

A little history

- **1970** — E. F. Codd, *"A Relational Model of Data for Large Shared Data Banks."*
- Radical idea: store data as simple **tables**, backed by **mathematical relations**.
- Hide storage details; query with a **declarative** language.
- Won over earlier **hierarchical** and **network** models.

A little history

- **1970** — E. F. Codd, *"A Relational Model of Data for Large Shared Data Banks."*
- Radical idea: store data as simple **tables**, backed by **mathematical relations**.
- Hide storage details; query with a **declarative** language.
- Won over earlier **hierarchical** and **network** models.
- Codd received the **Turing Award** (1981). The model still dominates today.

Why the relational model won

- **Simple** — everyone understands a table.

Why the relational model won

- **Simple** — everyone understands a table.
- **Formal** — grounded in set theory & logic → provable query rewriting.

Why the relational model won

- **Simple** — everyone understands a table.
- **Formal** — grounded in set theory & logic → provable query rewriting.
- **Declarative** — say *what*, not *how*; the system optimizes.

Why the relational model won

- **Simple** — everyone understands a table.
- **Formal** — grounded in set theory & logic → provable query rewriting.
- **Declarative** — say *what*, not *how*; the system optimizes.
- **Data independence** — physical storage can change freely.

Why the relational model won

- **Simple** — everyone understands a table.
- **Formal** — grounded in set theory & logic → provable query rewriting.
- **Declarative** — say *what*, not *how*; the system optimizes.
- **Data independence** — physical storage can change freely.
- **Powerful constraints** — integrity enforced by the system.

The big picture

A relational database is a set of relations.
A relation is a table.

Everything else — keys, constraints, SQL, joins — builds on this one idea.

Relations, Tuples, Attributes

A relation is a table

Example relation **`instructor`**:

id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	95000
22222	Einstein	Physics	95000
45565	Katz	Comp. Sci.	75000

A relation is a table

Example relation **`instructor`**:

id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	95000
22222	Einstein	Physics	95000
45565	Katz	Comp. Sci.	75000

- Each **row** = a **tuple**.

A relation is a table

Example relation **`instructor`**:

id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	95000
22222	Einstein	Physics	95000
45565	Katz	Comp. Sci.	75000

- Each **row** = a **tuple**.
- Each **column** = an **attribute**.

A relation is a table

Example relation **`instructor`**:

id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	95000
22222	Einstein	Physics	95000
45565	Katz	Comp. Sci.	75000

- Each **row** = a **tuple**.
- Each **column** = an **attribute**.
- The table as a whole = a **relation**.

Terminology map

Formal term	Table term	Example
Relation	Table	`instructor`
Tuple	Row	(10101, Srinivasan, Comp. Sci., 95000)
Attribute	Column	`salary`
Domain	Column's type	integers ≥ 0
Cardinality	# of rows	3
Degree / arity	# of columns	4

Attributes and domains

- Each attribute `A` draws values from a **domain** `dom(A)`.

Attributes and domains

- Each attribute `A` draws values from a **domain** `dom(A)`.
  - `salary` $\rightarrow$ non-negative numbers; `name` $\rightarrow$ strings.

Attributes and domains

- Each attribute `A` draws values from a **domain** `dom(A)`.
  - `salary` → non-negative numbers; `name` → strings.
- Domains are (in the pure model) **atomic** — indivisible.

Attributes and domains

- Each attribute A draws values from a **domain** `dom(A)`.
  - `salary` → non-negative numbers; `name` → strings.
- Domains are (in the pure model) **atomic** — indivisible.
  - No ”list of phone numbers” in a single cell (that’s **First Normal Form**).

Attributes and domains

- Each attribute **A** draws values from a **domain** `dom(A)`.
  - `salary` → non-negative numbers; `name` → strings.
- Domains are (in the pure model) **atomic** — indivisible.
  - No ”list of phone numbers” in a single cell (that’s **First Normal Form**).
- The special value **NULL** belongs to every domain — ”unknown / not applicable.”

Tuples

- A **tuple** is one row: an assignment of a value to each attribute.

Tuples

- A **tuple** is one row: an assignment of a value to each attribute.
- Written `t = (10101, 'Srinivasan', 'Comp. Sci.', 95000)`.

Tuples

- A **tuple** is one row: an assignment of a value to each attribute.
- Written `t = (10101, 'Srinivasan', 'Comp. Sci.', 95000)`.
- `t[name]` = `'Srinivasan'` (attribute access).

Tuples

- A **tuple** is one row: an assignment of a value to each attribute.
- Written `t = (10101, 'Srinivasan', 'Comp. Sci.', 95000)`.
- `t[name]` = `'Srinivasan'` (attribute access).
- Order of attributes is a convention; what matters is the **attribute name**.

A relation is a *set* of tuples

- Mathematically, a relation is a **set** — so:

A relation is a *set* of tuples

- Mathematically, a relation is a **set** — so:
  - **no duplicate tuples**, and

A relation is a *set* of tuples

- Mathematically, a relation is a **set** — so:
  - **no duplicate tuples**, and
  - **no inherent order** of rows.

A relation is a *set* of tuples

- Mathematically, a relation is a **set** — so:
  - **no duplicate tuples**, and
  - **no inherent order** of rows.
- Consequence: to identify a row, you need values that make it unique — a **key**.

A relation is a *set* of tuples

- Mathematically, a relation is a **set** — so:
  - **no duplicate tuples**, and
  - **no inherent order** of rows.
- Consequence: to identify a row, you need values that make it unique — a **key**.
- SQL relaxes this: real tables are *bags* (multisets) and may hold duplicate rows unless a key forbids it. We'll see this tension often.

Schema vs Instance

Relation schema

- The **schema** describes a relation's structure:

```
instructor(id, name, dept_name, salary)
```

Relation schema

- The **schema** describes a relation's structure:
  `instructor(id, name, dept_name, salary)`
- Names + domains + constraints. It is the **design**, and it rarely changes.

Relation schema

- The **schema** describes a relation's structure:
  `instructor(id, name, dept_name, salary)`
- Names + domains + constraints. It is the **design**, and it rarely changes.
- Written more fully:
  `instructor(id: char, name: varchar, dept_name: varchar, salary: numeric)`

Relation instance

- The **instance** is the actual set of tuples **right now**.

Relation instance

- The **instance** is the actual set of tuples **right now**.
- It changes with every INSERT/UPDATE/DELETE.

Relation instance

- The **instance** is the actual set of tuples **right now**.
- It changes with every INSERT/UPDATE/DELETE.
- The schema constrains which instances are **legal**.

Relation instance

- The **instance** is the actual set of tuples **right now**.
- It changes with every INSERT/UPDATE/DELETE.
- The schema constrains which instances are **legal**.

Relation instance

- The **instance** is the actual set of tuples **right now**.
- It changes with every INSERT/UPDATE/DELETE.
- The schema constrains which instances are **legal**.

Schema : Instance :: Class : Object :: Type : Value

Database schema vs database instance

- **Database schema** = the set of all relation schemas + constraints.

Database schema vs database instance

- **Database schema** = the set of all relation schemas + constraints.
- **Database instance** = a snapshot of all relation instances that satisfies the schema.

Database schema vs database instance

- **Database schema** = the set of all relation schemas + constraints.
- **Database instance** = a snapshot of all relation instances that satisfies the schema.
- The DBMS's job: only ever allow **legal** instances (constraints hold).

Keys

Keys — the central idea

- A **key** is a set of attributes that **uniquely identifies** a tuple.

Keys — the central idea

- A **key** is a set of attributes that **uniquely identifies** a tuple.
- Keys let us:

Keys — the central idea

- A **key** is a set of attributes that **uniquely identifies** a tuple.
- Keys let us:
  - refer to a specific row unambiguously,

Keys — the central idea

- A **key** is a set of attributes that **uniquely identifies** a tuple.
- Keys let us:
  - refer to a specific row unambiguously,
  - prevent duplicate/ambiguous data,

Keys — the central idea

- A **key** is a set of attributes that **uniquely identifies** a tuple.
- Keys let us:
  - refer to a specific row unambiguously,
  - prevent duplicate/ambiguous data,
  - connect tables together (foreign keys).

Keys — the central idea

- A **key** is a set of attributes that **uniquely identifies** a tuple.
- Keys let us:
  - refer to a specific row unambiguously,
  - prevent duplicate/ambiguous data,
  - connect tables together (foreign keys).
- Four flavors: **superkey, candidate key, primary key, foreign key**.

Superkey

- A **superkey** is any set of attributes whose values are unique across all legal instances.

Superkey

- A **superkey** is any set of attributes whose values are unique across all legal instances.
- For `instructor`:

Superkey

- A **superkey** is any set of attributes whose values are unique across all legal instances.
- For `instructor`:
  - `{id}` is a superkey.

Superkey

- A **superkey** is any set of attributes whose values are unique across all legal instances.
- For `instructor`:
  - `{id}` is a superkey.
  - `{id, name}` is also a superkey (adding attributes keeps uniqueness).

Superkey

- A **superkey** is any set of attributes whose values are unique across all legal instances.
- For `instructor`:
  - `{id}` is a superkey.
  - `{id, name}` is also a superkey (adding attributes keeps uniqueness).
- Superkeys may contain ”extra” attributes — they’re allowed to be redundant.

Candidate key

- A **candidate key** is a **minimal** superkey — remove any attribute and it is no longer unique.

Candidate key

- A **candidate key** is a **minimal** superkey — remove any attribute and it is no longer unique.
- `{id}` is a candidate key for `instructor`.

Candidate key

- A **candidate key** is a **minimal** superkey — remove any attribute and it is no longer unique.
- `{id}` is a candidate key for `instructor`.
- `{id, name}` is **not** — it's not minimal.

Candidate key

- A **candidate key** is a **minimal** superkey — remove any attribute and it is no longer unique.
- `{id}` is a candidate key for `instructor`.
- `{id, name}` is **not** — it's not minimal.
- A relation can have **several** candidate keys (e.g., a person's `national_id` and `email`).

Primary key

- The designer picks **one** candidate key as the **primary key (PK)**.

Primary key

- The designer picks **one** candidate key as the **primary key (PK)**.
- Conventions: choose something **small, stable, never-NULL**.

Primary key

- The designer picks **one** candidate key as the **primary key (PK)**.
- Conventions: choose something **small, stable, never-NULL**.
- Underlined in schema notation: id.

Primary key

- The designer picks **one** candidate key as the **primary key (PK)**.
- Conventions: choose something **small, stable, never-NULL**.
- Underlined in schema notation: id.
- The PK is how other tables will refer to this one.

Primary key

- The designer picks **one** candidate key as the **primary key (PK)**.
- Conventions: choose something **small, stable, never-NULL**.
- Underlined in schema notation: id.
- The PK is how other tables will refer to this one.

Primary key

- The designer picks **one** candidate key as the **primary key (PK)**.
- Conventions: choose something **small, stable, never-NULL**.
- Underlined in schema notation: id.
- The PK is how other tables will refer to this one.

```
instructor(id, name, dept_name, salary)
```

Choosing a good primary key

- **Minimal & stable** — values shouldn't change (don't key on a mutable email).

Choosing a good primary key

- **Minimal & stable** — values shouldn't change (don't key on a mutable email).
- **Never NULL** — a PK must always identify a row.

Choosing a good primary key

- **Minimal & stable** — values shouldn't change (don't key on a mutable email).
- **Never NULL** — a PK must always identify a row.
- **Natural vs surrogate:**

Choosing a good primary key

- **Minimal & stable** — values shouldn't change (don't key on a mutable email).
- **Never NULL** — a PK must always identify a row.
- **Natural vs surrogate:**
  - *natural* key = real-world attribute (student roll number).

Choosing a good primary key

- **Minimal & stable** — values shouldn't change (don't key on a mutable email).
- **Never NULL** — a PK must always identify a row.
- **Natural vs surrogate:**
  - *natural* key = real-world attribute (student roll number).
  - *surrogate* key = system-generated id (auto-increment) — common in practice.

Foreign key — connecting tables

- A **foreign key (FK)** is an attribute set in one relation that **references the primary key** of another (or the same) relation.

Foreign key — connecting tables

- A **foreign key (FK)** is an attribute set in one relation that **references the primary key** of another (or the same) relation.
- Example: `student.dept_name` references `department.dept_name`.

Foreign key — connecting tables

- A **foreign key (FK)** is an attribute set in one relation that **references the primary key** of another (or the same) relation.
- Example: `student.dept_name` references `department.dept_name`.
- The **referencing** relation: `student`. The **referenced** relation: `department`.

Foreign key — connecting tables

- A **foreign key (FK)** is an attribute set in one relation that **references the primary key** of another (or the same) relation.
- Example: `student.dept_name` references `department.dept_name`.
- The **referencing** relation: `student`. The **referenced** relation: `department`.
- FKs are how the relational model represents **relationships**.

Foreign key example

```
department(dept_name, building, budget)

student(id, name, dept_name, tot_cred)
                  |
                  +--> references department(dept_name)
```

Foreign key example

```
department(dept_name, building, budget)

student(id, name, dept_name, tot_cred)
                  |
                  +--> references department(dept_name)
```

- Every `student.dept_name` must match some existing `department.dept_name` (or be NULL, if allowed).

Integrity Constraints

Integrity constraints

Rules every legal instance must satisfy. Three core kinds:

1. **Domain constraints** — values come from the right domain / pass CHECKs.

Integrity constraints

Rules every legal instance must satisfy. Three core kinds:

1. **Domain constraints** — values come from the right domain / pass CHECKs.
2. **Entity integrity** — primary key attributes are never NULL.

Integrity constraints

Rules every legal instance must satisfy. Three core kinds:

1. **Domain constraints** — values come from the right domain / pass CHECKs.
2. **Entity integrity** — primary key attributes are never NULL.
3. **Referential integrity** — foreign keys reference existing rows.

Integrity constraints

Rules every legal instance must satisfy. Three core kinds:

1. **Domain constraints** — values come from the right domain / pass CHECKs.
2. **Entity integrity** — primary key attributes are never NULL.
3. **Referential integrity** — foreign keys reference existing rows.

Integrity constraints

Rules every legal instance must satisfy. Three core kinds:

1. **Domain constraints** — values come from the right domain / pass CHECKs.
2. **Entity integrity** — primary key attributes are never NULL.
3. **Referential integrity** — foreign keys reference existing rows.

The DBMS **enforces** these automatically.

Domain constraints

- Restrict the allowed values of an attribute.

Domain constraints

- Restrict the allowed values of an attribute.
- Type: `salary` is numeric.

Domain constraints

- Restrict the allowed values of an attribute.
- Type: `salary` is numeric.
- Explicit CHECK: `salary > 0`, `semester IN ('Spring','Summer','Fall','Winter')`.

Domain constraints

- Restrict the allowed values of an attribute.
- Type: `salary` is numeric.
- Explicit CHECK: `salary > 0`, `semester IN ('Spring','Summer','Fall','Winter')`.

Domain constraints

- Restrict the allowed values of an attribute.
- Type: `salary` is numeric.
- Explicit CHECK: `salary > 0`, `semester IN ('Spring','Summer','Fall','Winter')`.

```
salary NUMERIC CHECK (salary > 0)
```

Domain constraints

- Restrict the allowed values of an attribute.
- Type: `salary` is numeric.
- Explicit CHECK: `salary > 0`, `semester IN ('Spring','Summer','Fall','Winter')`.

```
salary NUMERIC CHECK (salary > 0)
```

- Any INSERT/UPDATE violating a domain constraint is **rejected**.

Entity integrity

- **The primary key can never be NULL** (in whole or part).

Entity integrity

- **The primary key can never be NULL** (in whole or part).
- Why? A NULL PK couldn't identify the row — defeating its purpose.

Entity integrity

- **The primary key can never be NULL** (in whole or part).
- Why? A NULL PK couldn't identify the row — defeating its purpose.
- Corollary: PK values must be **unique** *and* **present**.

Referential integrity

- A foreign key must either:

Referential integrity

- A foreign key must either:
  - match an existing PK value in the referenced relation, or

Referential integrity

- A foreign key must either:
  - match an existing PK value in the referenced relation, or
  - be entirely NULL (if the FK column allows NULLs).

Referential integrity

- A foreign key must either:
  - match an existing PK value in the referenced relation, or
  - be entirely NULL (if the FK column allows NULLs).
- Prevents "**dangling references**" — a `takes` row for a non-existent student.

Referential integrity: what can go wrong

Consider `takes.id` $\rightarrow$ `student.id`. The DBMS must react to:

- **Insert** a `takes` row for a student that doesn't exist $\rightarrow$ **reject**.

Referential integrity: what can go wrong

Consider `takes.id` → `student.id`. The DBMS must react to:

- **Insert** a `takes` row for a student that doesn't exist → **reject**.
- **Delete** a `student` that still has `takes` rows → reject, cascade, or set NULL.

Referential integrity: what can go wrong

Consider `takes.id` → `student.id`. The DBMS must react to:

- **Insert** a `takes` row for a student that doesn't exist → **reject**.
- **Delete** a `student` that still has `takes` rows → reject, cascade, or set NULL.
- **Update** a `student.id` that is referenced → same choices.

Referential integrity: what can go wrong

Consider `takes.id` → `student.id`. The DBMS must react to:

- **Insert** a `takes` row for a student that doesn't exist → **reject**.
- **Delete** a `student` that still has `takes` rows → reject, cascade, or set NULL.
- **Update** a `student.id` that is referenced → same choices.

Referential integrity: what can go wrong

Consider `takes.id` → `student.id`. The DBMS must react to:

- **Insert** a `takes` row for a student that doesn't exist → **reject**.
- **Delete** a `student` that still has `takes` rows → reject, cascade, or set NULL.
- **Update** a `student.id` that is referenced → same choices.

SQL lets you specify: `ON DELETE CASCADE`, `ON DELETE SET NULL`, `RESTRICT`.

Summary — key takeaways

- A **relation** is a set of tuples over named, typed attributes.

Summary — key takeaways

- A **relation** is a set of tuples over named, typed attributes.
- **Schema** (design) vs **instance** (data now).

Summary — key takeaways

- A **relation** is a set of tuples over named, typed attributes.
- **Schema** (design) vs **instance** (data now).
- **Keys**: superkey $\supseteq$ candidate key $\rightarrow$ chosen **primary key**; **foreign keys** connect relations.

Summary — key takeaways

- A **relation** is a set of tuples over named, typed attributes.
- **Schema** (design) vs **instance** (data now).
- **Keys**: superkey $\supseteq$ candidate key $\rightarrow$ chosen **primary key**; **foreign keys** connect relations.
- **Integrity constraints** (domain, entity, referential) keep data valid — and the DBMS enforces them.

Before next lecture

- Optional reading: Silberschatz Ch. 2 (relational model).

Next time — Lecture 3: NULLs, the University Schema, and **Relational Algebra I**.

Questions?