Study Guide: Introduction to the Relational Model (Chapter 2 Deep Dive)

July 27, 2025

1 The Relationl Model

This study guide elaborates on key concepts from Chapter 2, "Introduction to the Relational Model", building upon the foundational knowledge typically covered in slides and delving deeper into the nuances explained in the textbook.

1.1 Fundamental Concepts of the Relational Model

The relational model organizes data into *tables*, also known as **relations**.

- Each table has *multiple columns*, referred to as **attributes**.
- Each column has a **unique name**.
- Each *row* of the table represents one piece of information, known as a **tuple**. A row in a table represents a relationship among a set of values.
- The **order of tuples in a relation is irrelevant**, as a relation is mathematically defined as a set of tuples. This means two tables with the same set of data are considered the same relation regardless of tuple order.
- Examples include the **instructor** relation with attributes (ID, name, dept_name, salary) and the **department** relation with (dept_name, building, budget).

1.2 Keys and Integrity Constraints

Keys are fundamental to uniquely identify tuples and ensure data integrity.

- **Superkey (K)**: A set of one or more attributes that, taken collectively, are **sufficient to identify a unique tuple** in each possible relation for a given schema.
  - **Example**: {ID} and {ID, name} are superkeys for the `instructor` relation because the ID alone is sufficient to distinguish instructors.
- **Candidate Key**: A superkey **K** that is **minimal**. This means no proper subset of **K** is also a superkey.
  - **Example**: ID is a candidate key for `instructor`, but {ID, name} is not, because ID alone is sufficient.
- **Primary Key**: One of the candidate keys is **selected to be the primary key**.
  - It is customary to list primary key attributes first in the relation schema and to **underline** them.
  - **Example**: In `department(dept_name, building, budget)`, dept_name is the primary key. In `classroom(building, room_number, capacity)`, {building, room_number} is the primary key.
- **Foreign Key Constraint**: A value in one relation (**referencing relation**) must appear in another relation (**referenced relation**).
  - Typically, a foreign key in the referencing relation refers to the **primary key** of the referenced relation.

- **Example**: `dept_name` in `instructor` is a foreign key referencing `department`. This ensures an instructor can only be assigned to an existing department.
- In SQL, foreign keys are specified using the `foreign key` clause in `create table` statements.

- **Referential Integrity Constraint**: A more general type of constraint than a foreign key constraint, where the referenced attributes do not necessarily form a primary key of the referenced relation.

1.3 Schema Diagrams

Schema diagrams provide a **visual representation** of database structure, including relations, attributes, and key constraints.

- Each **relation is depicted as a box**, with the relation name at the top.
- Attributes are listed inside the box.
- **Primary key attributes are underlined**.
- **Foreign key constraints** are shown as **single-headed arrows** from the foreign-key attributes in the referencing relation to the primary key of the referenced relation.
- **Referential integrity constraints** (that are not foreign-key constraints) are shown with a **two-headed arrow**.
- **Example**: The schema diagram for the university database visually represents relations like `instructor`, `department`, `student`, `takes`, `teaches`, etc., and their interrelationships.

1.4 Relational Algebra Operations

Relational algebra is a **formal, functional query language** that forms the **theoretical basis for SQL**. It consists of operations that take one or two relations as input and produce a new relation as a result.

1.4.1 Unary Operations (operate on a single relation)

- **Selection (σ)**:
  - Selects **tuples (rows)** from a relation that satisfy a given **predicate**.
  - **Notation**: $\sigma_{\text{predicate}}(\text{Relation})$. The predicate appears as a subscript.
  - Predicates can use comparison operators ($=, \neq, <, \leq, >, \geq$) and logical connectives (AND ($\wedge$), OR ($\vee$), NOT ($\neg$)).
  - **Example**: Find all instructors in the "Physics" department: $\sigma_{\text{dept\_name} = \text{"Physics"}}(\text{instructor})$. The result is shown in Figure 2.10.
  - **Example**: Find instructors in Physics with salary over \$90,000: $\sigma_{\text{dept\_name} = \text{"Physics"} \wedge \text{salary} > 90000}(\text{instructor})$.
- **Projection (Π)**:
  - Selects **specific attributes (columns)** from a relation, creating a new relation with a subset of the original attributes.
  - **Notation**: $\Pi_{\text{AttributeList}}(\text{Relation})$.
  - In formal relational algebra, this operation **implicitly eliminates duplicate tuples** from the result.
  - **Example**: List instructors' ID, name, and salary: $\Pi_{\text{ID, name, salary}}(\text{instructor})$. The result is shown in Figure 2.11.
- **Rename (ρ)**:
  - Used to **change the name of a relation or its attributes**.
  - It's particularly useful when performing operations that require comparing tuples within the *same* relation, by creating aliases to avoid ambiguity.
  - While not explicitly detailed in the slides, it's one of the 6 basic operations.

1.4.2 Binary Operations (operate on two relations)

- **Cartesian Product ($\times$)**:
  - Combines **every tuple from the first relation with every tuple from the second relation**.
  - The schema of the result includes all attributes from both relations. If attributes have the same name, they are prefixed with the relation name (e.g., `instructor.ID`, `teaches.ID`).
  - **Example**: `instructor` $\times$ `teaches`.
  - This operation often generates **very large intermediate relations** containing many irrelevant tuples that do not logically correspond. It forms the basis for the join operation.
- **Join ($\bowtie$)**:
  - A join operation $r \bowtie_\theta s$ is formally defined as a **Cartesian product followed by a selection operation** on a predicate θ. $r \bowtie_\theta s = \sigma_\theta(r \times s)$.
  - This filters the Cartesian product to include only logically related tuples.
  - **Example**: To find tuples from (`instructor` $\times$ `teaches`) that pertain to instructors and the courses they taught, we use: $\sigma_{\text{instructor.ID = teaches.ID}}(\text{instructor} \times \text{teaches})$. This is equivalent to `instructor` $\bowtie_{\text{instructor.ID = teaches.ID}}$ `teaches`. The result is shown in Figure 2.13 and Figure 3.7.
- **Set Operations (Union ($\cup$), Intersection ($\cap$), Set Difference ($-$))**:
  - These operations are analogous to set theory operations.
  - They require relations to be **union compatible**: they must have the **same number of attributes** and corresponding attributes must have **compatible data types**.
  - **Union ($\cup$)**: Combines all tuples from two relations. In formal relational algebra, **duplicate tuples are implicitly eliminated**.
  - **Example**: Courses taught in Fall 2017 OR Spring 2018: $\Pi_{\text{course id}}(\sigma_{\text{semester = "Fall"} \wedge \text{year=2017}}(\text{section})) \cup \Pi_{\text{course id}}(\sigma_{\text{semester = "Spring"} \wedge \text{year=2018}}(\text{section}))$.
  - **Intersection ($\cap$)**: Returns only those tuples that appear in **both** relations. Duplicates are implicitly eliminated.
  - **Example**: Courses taught in BOTH Fall 2017 AND Spring 2018: $\Pi_{\text{course id}}(\sigma_{\text{semester = "Fall"} \wedge \text{year=2017}}(\text{section})) \cap \Pi_{\text{course id}}(\sigma_{\text{semester = "Spring"} \wedge \text{year=2018}}(\text{section}))$.
  - **Set Difference ($-$)**: Returns tuples that are in the first relation **but not** in the second. $r - s$ produces tuples in r but not in s.
  - **Example**: Courses taught in Fall 2017 BUT NOT in Spring 2018: $\Pi_{\text{course id}}(\sigma_{\text{semester = "Fall"} \wedge \text{year=2017}}(\text{section})) - \Pi_{\text{course id}}(\sigma_{\text{semester = "Spring"} \wedge \text{year=2018}}(\text{section}))$.

1.4.3 Assignment Operation ($\leftarrow$)

- Allows the result of a relational algebra expression to be **assigned to a temporary relation variable**.
- This is useful for breaking down complex queries into smaller, more manageable steps.
- **Example**: `courses_fall_2017` $\leftarrow \Pi_{\text{course id}}(\sigma_{\text{semester = "Fall"} \wedge \text{year=2017}}(\text{section}))$. This temporary relation can then be used in subsequent expressions.

1.5 Duplicate Handling: Relational Algebra (Formal) vs. SQL (Practical)

This is a crucial distinction that often causes confusion between the theoretical model and practical implementations.

- **Formal Relational Algebra**:
  - Relations are defined as **sets of tuples**.
  - By definition, **sets cannot contain duplicate elements**.
  - Therefore, operations like Projection (Π) and Union ($\cup$) **implicitly eliminate duplicates** from their results.

- **Practical SQL Implementations**:
  - SQL **tables (relations) can contain duplicate tuples** by default, unless specific constraints (like primary key or 'UNIQUE') prevent it.
  - To explicitly remove duplicates in a `SELECT` query, the `DISTINCT` keyword is used (e.g., `SELECT DISTINCT dept_name FROM instructor;`).
  - For set operations:
    - `UNION` removes duplicates by default, mirroring formal relational algebra.
    - `UNION ALL` retains all duplicate tuples.
    - `INTERSECT` removes duplicates by default.
    - `INTERSECT ALL` retains duplicates, with the number of duplicates equal to the minimum count in both input relations.
    - `EXCEPT` (or `MINUS` in some systems like Oracle) removes duplicates by default.
    - `EXCEPT ALL` retains duplicates, with the count equal to the difference in duplicate counts between the first and second relations.
  - This behavior in SQL is often described using the concept of **multisets** (or bags), which are collections that allow duplicate elements. (Note: This is external information to the provided sources, but helps to frame the difference).

1.6 Equivalent Queries and Query Optimization

There are often **multiple ways to write the same query** in relational algebra or SQL, all yielding the same result. However, the **efficiency of execution can vary drastically** depending on the order of operations.

- **Optimization Principle: Push Selections Early**:
  - A key optimization strategy is to **apply selection (filtering) operations as early as possible** in the query execution plan.
  - This **reduces the size of intermediate results**, leading to more efficient processing, especially for subsequent operations like joins.
  - **Example**: Consider finding information about courses taught by instructors in the Physics department.
    - **Less efficient**: $\sigma_{\text{dept\_name = "Physics"}}(\text{instructor} \bowtie_{\text{instructor.ID = teaches.ID}} \text{teaches})$. Here, the join is computed first, potentially creating a large intermediate table, and *then* filtered.
    - **More efficient**: $(\sigma_{\text{dept\_name = "Physics"}}(\text{instructor})) \bowtie_{\text{instructor.ID = teaches.ID}} \text{teaches}$. Here, the `instructor` relation is filtered *before* the join, significantly reducing the number of tuples involved in the join operation.
  - Database systems have **query optimizers** that attempt to transform user queries into more efficient execution plans, often by applying rules like pushing selections. They consider various equivalent expressions and estimate their costs.
- **Join Ordering**:
  - The order of join operations can also significantly affect performance.
  - Relational join operations are **associative** ($(r_1 \bowtie r_2) \bowtie r_3 \equiv r_1 \bowtie (r_2 \bowtie r_3)$) and **commutative** ($r_1 \bowtie r_2 \equiv r_2 \bowtie r_1$).
  - Optimizers use these properties to reorder joins to minimize intermediate result sizes.