The Relational Model and Integrity Constraints

Table of contents

Outline . . . 1
Relational model concepts . . . 1
Question — Schema and Instance . . . 3
Integrity constraints . . . 4
Question — Key . . . 5
Question — Key (multiple candidate keys) . . . 5
Exercise — name an example of each . . . 7
Question — Integrity Constraints (password) . . . 7
Question — Integrity Constraints (which is violated?) . . . 8
Constraints and operations . . . 8
The transaction concept . . . 10

Module 2, part 1. See relational-mapping-notation[1] for the schema notation used here and in 2026-03-16-er-to-relational-mapping[2].

Outline

- Relational model concepts — relations, attributes, domains, tuples
- Integrity constraints
- The transaction concept

Relational model concepts

Introduced by E.F. Codd in 1970. Many DBMS products are based on this model — a sound theoretical foundation with a simple, uniform data structure called a **relation**. Four basic concepts: **relations**, **attributes**, **domains**, **tuples**.

[1] courses/infs1200/relational-mapping-notation.pdf
[2] courses/infs1200/2026-03-16-er-to-relational-mapping.pdf

Relations

A relation is the main construct for representing data — informally, a set of records, similar to a table with columns and rows. The term *table* is used interchangeably with *relation*, but **every relation is a table; not every table is a relation** — relations have specific properties based on mathematical set theory.

A counter-example is a spreadsheet-style sales report: a two-tier header (`City: Brisbane` / `Product` / `Year: 1998` above `Region` / `Suburb` / blank / `Qtr 1` … `Qtr 4`) over rows like `South | Algester | Disks | 32 | 243 | 23 | 246`. Merged and stacked header cells mean the columns are not a flat list of attributes, so it is a table but **not a relation**.

Relation components

Four of the model's terms sit on one picture: the **relation name** labels the whole table, the **attribute names** label the columns, each row is a **tuple**, and every value in a column is drawn from that attribute's **domain**.

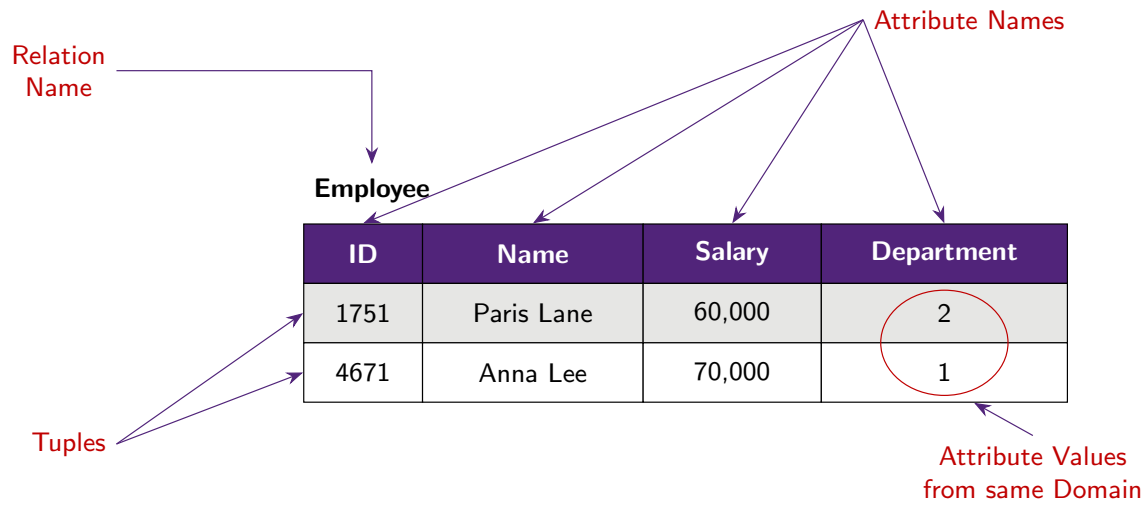

Figure 1: The anatomy of a relation, on an Employee example. Relation name above the table, attribute names in the purple header, tuples as rows, and one column circled: every value in it comes from the same domain.

Domains

A **domain** `D` is a set of atomic values — an atomic value is indivisible as far as the relational model is concerned. Each domain has a data type/format: integers, numbers/currency,

fixed/variable-length character strings, date/timestamp, a sub-range of a data type (e.g. `1 ≤ grade ≤ 7`), or an enumerated type (e.g. `Gender ∈ {Male, Female, Other}`) — including format-constrained domains like Australian phone numbers (`61` + 9 digits) or car registrations (6 alphanumeric characters, no `Q`).

Attributes

Each **attribute** `A` is the name of a role played by some domain `D` in a relation `R`. The number of attributes in `R` is its **degree**. Same-named attributes across relations don't necessarily share a domain (`Department.id` and `Employee.id` can be different domains despite the shared name `id`), and differently-named attributes can share a domain (`Employee.id` and `Employee.managerId` can both be drawn from the same "employee id" domain, despite the different names).

Tuples

Each **tuple** `t` is an ordered list of n values `t = <v1, v2, …, vn>`, where each value belongs to the corresponding attribute's domain, or is the special value `NULL`. `t` is called an n-tuple.

Relation schema and instance

- **Relation schema**: `R [A1, A2, …, An]` — a relation name `R` and its list of attributes; n is the **degree of the relation**. E.g. `Employee [id, name, sex, salary, department]` is a schema of degree 5.
- **Relation instance**: `r(R)`, a set of n-tuples `r = {t1, t2, …, tm}` — the actual data, at a point in time.

Question — Schema and Instance

Match each characteristic to Schema or Instance: "data in the database", "specified during database design", "data describing the data", "created through data update operations".

> **i** Solution
>
> **Instance**: "data in the database", "created through data update operations" (inserts/updates/deletes change the instance, not the schema). **Schema**: "specified during database design", "data describing the data" (the schema is metadata — it describes the shape data must take, not the data itself).

Ordering of tuples and values

Relations are *sets* of tuples — mathematically, a set has no implied order. Semantically (e.g. when writing queries), tuple order is irrelevant. Physically, tuples reside on blocks of secondary storage, which have a *partial* ordering, so at that level tuples do have an order — but two instances with the same tuples in a different order are still the *same* relation. Likewise, an n-tuple is syntactically an ordered list — every tuple in one relation must list values in the same attribute order — but semantically, which order is chosen doesn't matter, as long as the attribute/value correspondence is maintained consistently.

Integrity constraints

Integrity constraints are rules that enforce the *correctness* of a database — they must hold on every instance of the schema. Five kinds: **domain**, **key**, **entity integrity**, **referential integrity**, **semantic**.

Domain constraint

A domain constraint violation occurs when an attribute's value doesn't appear in its corresponding domain. Assuming the domain of `Employee.id` is a 4-digit integer, inserting `("LOL", Anna Lee, 70,000, 4)` is rejected; `(4671, Anna Lee, 70,000, 4)` is accepted.

Key

A **key** is a minimal set of attributes that uniquely identifies tuples in a relation — *minimal* meaning no redundant attributes, not necessarily the smallest possible set. A schema can have more than one key (each a **candidate key**); the one chosen as the relation's main key is the **primary key** (conventionally underlined). A **key constraint** violation occurs when a tuple is inserted/modified to share a key value with an existing tuple. With `Employee` holding `(1751, Paris Lane, 60,000, 2)` and `(4671, Anna Lee, 70,000, 1)`, inserting `(4671, Ben Cho, 70,000, 4)` is rejected because `id = 4671` is taken; `(1023, Ben Cho, 70,000, 4)` is accepted.

See relational-keys[3] for the minimality argument and for the two key questions below drawn as one picture.

[3] courses/infs1200/relational-keys.pdf

Question — Key

Assuming department IDs are unique, which of the following is a key for `Department [id, name, manager]`*: (A)* `(id)`*, (B)* `(id, name)`*, (C)* `(id, manager)`*, (D) all of the above?*

> **Solution**
>
> **(A)** `(id)` — since IDs are already unique on their own, `(id)` is the *minimal* key; `(id, name)` and `(id, manager)` both contain a redundant extra attribute (adding anything to an already-unique set isn't minimal).

Question — Key (multiple candidate keys)

Assuming department IDs are unique ***and*** *the combination of Name and Manager is also unique per department, which of the following is a key(s): (A)* `(id)`*, (B)* `(id, name)`*, (C)* `(name, manager)`*, (D) both A and C?*

> **Solution**
>
> **(D) Both A and C.** `(id)` is still minimal and unique on its own; now `(name, manager)` is *also* minimal and unique, making it a second candidate key. `(id, name)` is still not minimal since `(id)` alone already suffices.

Entity integrity constraint

An entity integrity constraint violation occurs when a tuple is inserted/modified such that (any part of) its primary key is `NULL` — for a composite primary key, *no part* of it can be null. Inserting `(NULL, Jack Smith, 40,000, 1)` into `Employee` is rejected; `(2034, Jack Smith, 40,000, 1)` is accepted.

Foreign keys

A **foreign key** is a set of attributes in one relation that links it to another relation's primary key. Formally: let `FK` be attributes in `R1` and `PK` the primary key of `R2`. `FK` in `R1` is a foreign key referencing `PK` in `R2` if `FK`/`PK` share a domain, and for every tuple `t1` in `R1`, either `t1[FK]` is `NULL`, or some tuple `t2` in `R2` has `t1[FK] = t2[PK]`.

```
Department [id, name, manager]
Department.manager references Employee.id
```

```
Employee [id, name, sex, salary, department]
Employee.department references Department.id
```

Both directions are live at once here. In the example instance, `Department` 4 (Human Resources) has a `NULL` manager, employee 2034 has a `NULL` department, and both rows are legal — an absent reference is fine, a dangling one is not. The two figures in foreign-keys[4] draw this instance with the foreign-key arrows on it.

Self-referencing relations are also possible — a table can reference itself. For example, add a `ManagerID` column to `Employee` and declare `Employee.managerID references Employee.id`. **Composite foreign keys** are possible too, referencing a multi-attribute primary key:

```
Student [sid, name]

Course [cid, department, manager]

Enrollment [sid, cid, department, grade]
Enrollment.sid references Student.sid
Enrollment.{cid, department} references Course.{cid, department}
```

`Course` is keyed on the pair `(cid, department)`, so `Enrollment` has to carry both columns to name one course — one composite constraint over a tuple of columns, not two separate ones.

Referential integrity constraint

A referential integrity constraint violation occurs when a foreign key value doesn't match any existing primary key value in the referenced relation (and isn't `NULL`). The constraint *guarantees that a department with number 2 exists before the "Grace Mills" tuple is stored*: `(2670, Grace Mills, 50,000, 5)` is rejected because there is no `Department` 5, while `(2670, Grace Mills, 50,000, 2)` is accepted.

Semantic (business-rule) constraint

Semantic constraints are generally defined by the business/organisation **during client consultation** (not derivable from the schema's structure alone) — e.g. "the salary of an employee should not exceed the employee's supervisor's salary", or "the maximum number of hours that an employee can work on a project is 56". Often implemented in a constraint specification language (SQL) using **triggers and assertions**.

[4] courses/infs1200/foreign-keys.pdf

Worked on the example instance: department 1's manager is Anna Lee, who earns 70,000, so inserting `(2967, Arron Dills, 100,000, 1)` breaks the salary rule and is rejected; `(2967, Arron Dills, 40,000, 1)` is accepted. Nothing about the *shape* of the two relations rules the first one out — that is exactly what makes it semantic rather than structural.

Exercise — name an example of each

Given the schema below, give an example of (1) a key, (2) a foreign key, (3) a domain constraint.

```
Student [stID, name, email]

Enrolment [stID, cCode, sem, year]
Enrolment.stID references Student.stID
Enrolment.cCode references Course.cCode

Course [cCode, title, units]
```

i Solution

1. **Key** — `stID` in relation `Student`.
2. **Foreign key** — `stID` in relation `Enrolment`.
3. **Domain constraint** — `cCode`, four letters followed by four numbers.

The same attribute name supplies the first two answers depending on which relation it is read in: it is a key where it identifies, a foreign key where it refers.

Question — Integrity Constraints (password)

Opening an online bank account, you enter your usual password "password123", but get: "your password must contain at least one capital letter and a number". What kind of constraint is this?

i Solution

Domain constraint — the domain of "password" is defined as strings matching a particular format (1 capital, 1 digit); "password123" simply isn't a member of that domain.

Question — Integrity Constraints (which is violated?)

Given `Department [id, name, manager]` *and* `Employee [id, name, salary, department]`*, with* `Department` *rows* `(1, Marketing, 4671)`*,* `(2, Development, 1751)`*, and* `Employee` *rows* `(1751, Paris Lane, 60000, 2)`*,* `(4671, Anna Lee, 70000, 1)`*,* `(2670, Grace Mills, 50000, 2)`*,* `(2034, Jack Smith, 40000, 1)` *— which constraint is violated by each of the following?*

1. *Inserting* `(2670, James Smith, 40000, 1)` *into* `Employee`*.*
2. *Inserting* `(2644, James, Smith, 1)` *into* `Employee` (note: only 4 values given for a 4-attribute relation, but shifted).
3. *Inserting* `(2644, James Smith, 40000, 3)` *into* `Employee`*.*
4. *Deleting* `(2, Development, 1751)` *from* `Department`*.*
5. *Updating* `(2, Development, 1751)` *to* `(2, Development, 2034)` *in* `Department`*.*

i Solution

1. **Key constraint** — `id = 2670` already exists in `Employee`.
2. **Domain constraint** — the shift means `salary` receives a non-numeric value (`"Smith"`), which isn't a member of `salary`'s domain.
3. **Referential integrity constraint** — no `Department` with `id = 3` exists.
4. **Referential integrity constraint** — `Employee` rows with `department = 2` still exist (Grace Mills), so deleting `Department 2` would leave a dangling foreign key.
5. **None of the above (by the *stated* constraints)** — `Employee` with `id = 2034` doesn't actually work in `Department 1`, but there's no *declared* constraint stopping a department's manager from being an employee of a different department; this would only be a semantic constraint if the business rule "a manager must work in the department they manage" were explicitly specified.

Constraints and operations

Enforcing integrity constraints keeps the database consistent — insert, modify, and delete operations must not leave the database in an inconsistent state; if a database update submitted to the DBMS would violate integrity, **it must be rejected**.

Insertion and deletion differ in which constraints they can violate:

- **Insertion/modification** can violate any of the five constraint types (domain, key, entity integrity, referential integrity, semantic).
- **Deletion** can only violate **referential** or **semantic** constraints. A referential integrity violation on delete can be handled by rejecting the delete, cascading it (deleting dependent rows too), or modifying the referencing attribute values to a default value.

Employee

ID	Name	Salary	Department
1751	Paris Lane	60,000	2
4671	Anna Lee	70,000	1
1023	Ben Cho	70,000	4
2034	Jack Smith	40,000	1
2670	Grace Mills	50,000	2

Department

ID	Name	Manager
1	Marketing	4671
2	Development	1751
4	Human Resources	1023

Insertion or modification — can violate any of the five

LOL	**John Doe**	**70,000**	**4**	Domain constraint
4671	**John Doe**	**70,000**	**4**	Key constraint
NULL	**John Doe**	**70,000**	**4**	Entity integrity constraint
1111	**John Doe**	**70,000**	**6**	Referential integrity constraint
1111	**John Doe**	**99,999**	**4**	Semantic integrity constraint

Deletion — can violate only these two

Removing the department with ID = 2	Referential integrity constraint
Removing the employee with ID = 1023, assuming every department needs at least one employee	Semantic integrity constraint

Figure 2: Which operation can violate which constraint, on an Employee/Department instance. Each of the five candidate insertions differs from a legal row in exactly one respect; the two deletion cases sit below them.

Read the five candidate rows as a controlled experiment: each differs from a legal row in exactly one respect, and that one difference names the constraint. The last one is the giveaway that semantic constraints are not structural — `(1111, John Doe, 99,999, 4)` is a perfectly well-formed tuple with a fresh key, a real department and no nulls, and is still rejected. It breaks the salary rule: 99,999 exceeds the 70,000 earned by Ben Cho, the manager of department 4.

The transaction concept

A **transaction** is an executing program that includes database operations (reads, inserts, deletes, updates). At the end of a transaction, the database must be left in a valid/consistent state satisfying all constraints — but constraint violations are allowed at *intermediate* steps within the transaction. That is the whole point: transactions allow a suite of queries to run where constraint violations in intermediate steps are permitted.

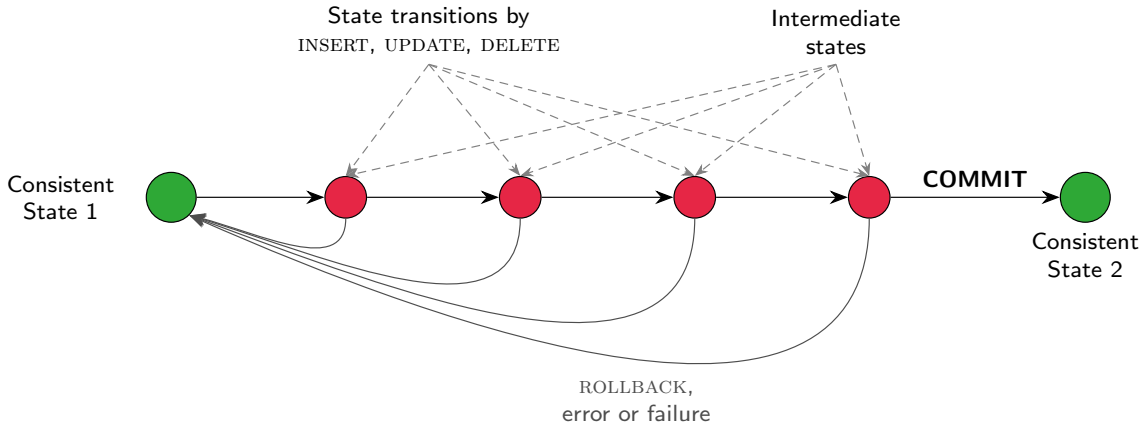

Figure 3: A transaction as a walk between consistent states. INSERT/UPDATE/DELETE move the database through intermediate states that may violate constraints; COMMIT is legal only into a consistent state, and ROLLBACK (or an error or failure) returns to the state the transaction started from. INFS1200 has no house notation for transaction diagrams; the shapes and colours here follow the figure in the SAP MaxDB documentation, not a course convention.

Example: given `Department [id, name, manager]` and `Employee [id, name, sex, salary, department]`, with the business rules "every department must have 1 employee" and "every employee must work for a department" — neither a new department (no employees yet) nor a new employee (no department yet) can be inserted alone without violating one of these rules. A **transaction** that inserts both the new department and its first employee together

resolves this: the intermediate state (after just one insert) is inconsistent, but the final state (after both inserts) is valid.