Update Operations and Constraint Violations

Table of contents

Which operation can violate what . . . 1
Handling a referential violation on delete . . . 3
Worked diagnoses . . . 4
Failures cascade . . . 4
The transaction concept . . . 5

There are three update operations on a relational database — **insert**, **delete**, and **modify** — and each can break a different subset of the integrity-constraints[1]. Knowing which subset turns "does this operation violate anything?" from a search into a checklist. Introduced in 2026-03-09-the-relational-model-and-integrity-constraints[2], drilled across week4-tutorial-applied-class-3-relational-integrity[3] and week6-tutorial-case-study-4-easydrive-insurance[4]. This is the model-level account; sql-dml[5] is the same three operations as SQL statements.

Which operation can violate what

	Domain	Key	Entity integrity	Referential integrity	Semantic
Insert					
Delete					
Modify					

Insert can violate all five: it supplies fresh values for every attribute, so any of them can be out of domain, duplicate an existing key, leave a primary key component null, point at a nonexistent row, or break a business rule. Five candidate tuples for the same two-table schema, differing in one value each, show one of each:

[1] courses/infs1200/integrity-constraints.pdf
[2] courses/infs1200/2026-03-09-the-relational-model-and-integrity-constraints.pdf
[3] courses/infs1200/week4-tutorial-applied-class-3-relational-integrity.pdf
[4] courses/infs1200/week6-tutorial-case-study-4-easydrive-insurance.pdf
[5] courses/infs1200/sql-dml.pdf

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

LOL	**John Doe**	**70,000**	**4**	⊗	Domain constraints
4671	**John Doe**	**70,000**	**4**	⊗	Key constraints
NULL	**John Doe**	**70,000**	**4**	⊗	Entity Integrity constraints
1111	**John Doe**	**70,000**	**6**	⊗	Referential Integrity constraints
1111	**John Doe**	**99,999**	**4**	⊗	Semantic Integrity constraints

Figure 1: One insert, five ways to break it. Each candidate row differs from a legal `Employee` tuple in exactly one value, and each trips a different constraint: `LOL` is not a four-digit integer (domain), `4671` already exists (key), a null `ID` is a null primary key (entity integrity), department `6` is not in `Department` (referential integrity), and `99,999` exceeds the supervisor's salary, which the business rule forbids (semantic). The crimson outline plus ⊗ badge is the course's notation for an illegal tuple.

Modification is the same picture: it can produce any of those five rows out of a legal one, so it inherits the whole column.

Delete is the interesting row. It supplies no values at all, so it cannot possibly produce an out-of-domain value, a duplicate key, or a null primary key. What it *can* do is remove a row that something else still references — a referential integrity violation on the *other* relation — or break a business rule. Both happen on the same two-table `Employee`/`Department` schema: deleting the department with `id = 2` violates **referential integrity**, because employees still sit in it, while deleting employee `1023` violates a **semantic** constraint if the rule "every department must have at least one employee" has been stated. This asymmetry is worth remembering: for a delete, the only question is who points at the row being deleted.

Modify behaves like an insert restricted to the columns it touches: changing a key column risks a key violation, changing a foreign key risks a referential one, and changing a primary key component to `NULL` risks entity integrity.

Handling a referential violation on delete

Rejecting the operation isn't the only option. At the relational-model stage there are exactly three outcomes for a referential integrity violation — it can be:

Policy	Effect
Rejected	Refuse the delete; the referencing rows keep the referenced row alive
Cascaded	Delete the dependent rows too
Modified to a default value	Keep the dependent rows, but reset the referencing attribute's value

That three-way choice is what is examinable *here*; the finer-grained named actions — `RESTRICT` (the default), `CASCADE`, `SET NULL` and `SET DEFAULT`, each available on both delete and update — arrive with SQL DDL and are examined there, not at the model level. See sql-ddl[6] for their exact semantics and syntax, and foreign-keys[7] for the constraint itself.

The choice is a property of the foreign key, decided at design time from what the dependent rows *mean* without the row they reference: an insurance claim makes no sense without its policy, so EasyDrive declares its claims `ON DELETE CASCADE`.

[6] courses/infs1200/sql-ddl.pdf
[7] courses/infs1200/foreign-keys.pdf

Worked diagnoses

Against the company schema `Employee [ssn, …, superSSN, dNumber]`, `Department [dName, dNumber, mgrSSN, mgrStartDate]`, `DeptLocations [dNumber, dLocation]`, `Project [pName, pNumber, pLocation, dNum]`, `WorksOn [essn, pNo, hours]`, `Dependent [essn, depName, …]`, plus two business rules — an employee's salary cannot exceed their supervisor's, and a department cannot have more than three active projects:

Operation	Violates
Insert an employee with a fresh `ssn`, a real supervisor, salary below the supervisor's	**Nothing** — a well-formed insert
Insert `('ProductA', 'A', 'Bellaire', 2)` into `Project` when no department 2 exists	Referential integrity **and** domain (`pNumber` is an integer, not `'A'`)
Insert a department whose `dNumber` already exists and whose `mgrSSN` doesn't	Key **and** referential integrity
Insert `(677678989, null, 40.0)` into `WorksOn`, `essn` not an employee	Entity integrity (`pNo` is part of the key) **and** referential integrity
Insert an employee earning more than their supervisor	Semantic
Delete the `WorksOn` rows for one employee	**Nothing** — nothing references `WorksOn`, and no rule requires an employee to have assignments
Delete an `Employee` row	Referential integrity — `WorksOn`, `Dependent`, `Department` and `Employee` itself may all reference that `ssn`
Modify a department's `mgrSSN` to another existing employee	**Nothing**, absent a stated rule about who may manage a department
Modify an employee's `superSSN` to an `ssn` that doesn't exist	Referential integrity
Insert a fourth active project for a department	Semantic

Two habits fall out of this. First, always check whether the row being deleted is *referenced*, not just whether it exists. Second, "no violation" is a legitimate answer — several of the operations above are perfectly legal, and the temptation is to invent a rule that forbids them.

Failures cascade

A rejected operation changes nothing, which means later operations that assumed it succeeded fail too. In the EasyDrive case study, inserting a `Vehicle` with `VehiclePurpose = 'Personal'` fails on a domain constraint (the column is `ENUM('Private', 'Business')`); the *next* insert, a `Policy` referencing that vehicle, then fails on referential integrity even though its own values

look fine. When a sequence of operations is being analysed, each one has to be judged against the state the previous ones actually left behind — not the state they were intended to leave.

The transaction concept

A **transaction** is an executing program containing database operations — reads, inserts, deletes, updates. The consistency requirement applies at the *end* of a transaction, not between its individual steps: constraint violations are permitted at intermediate points, as long as the final state satisfies every constraint specified on the schema.

A transaction is a walk between two consistent states. It starts from consistent state 1, and each `INSERT`, `UPDATE` or `DELETE` moves the database through **intermediate states** that need not satisfy the constraints; a `COMMIT` ends the walk at consistent state 2, while a `ROLLBACK` — or any error or failure along the way — returns the database to consistent state 1 as though nothing had happened. Either way the database is never *left* inconsistent.

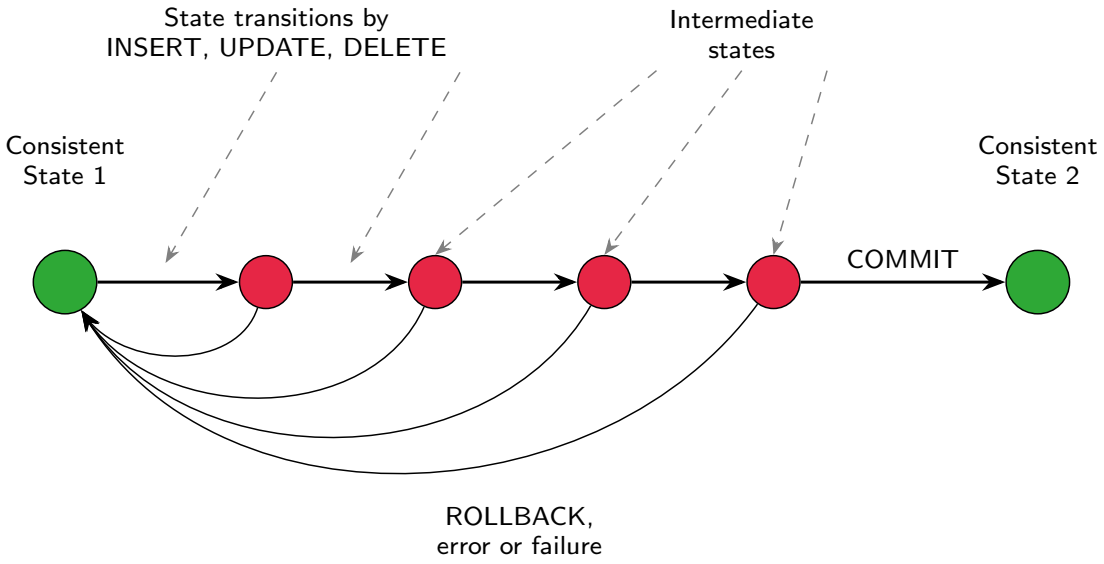


Figure 2: The transaction as a walk between consistent states. Green = a state satisfying every constraint, red = an intermediate state that need not. Only `COMMIT` and `ROLLBACK` end the walk, and both end it on a green state. The course has no house notation for transaction diagrams, so the shapes and colours here are not a convention to reproduce in an answer.

This is what makes mutually-dependent business rules satisfiable at all. Given "every department must have at least one 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

on its own without breaking a rule. A single transaction that inserts the department *and* its first employee resolves the deadlock: the intermediate state after the first insert is inconsistent, but the final state after both inserts is valid, and that is the state the constraints are judged against.