

Topic 08

University Computer Science

What this course is, and how to use it

This is the eighth course in University Computer Science: **databases** – how to model a domain as relations, query them precisely, keep them consistent under load, and govern who may see what.

Scope limit, stated once: a study course, not a degree; the teaching schema is an original university-enrolment database, and every query and output in this handout was executed against it by the course suite (SQLite, deterministic).

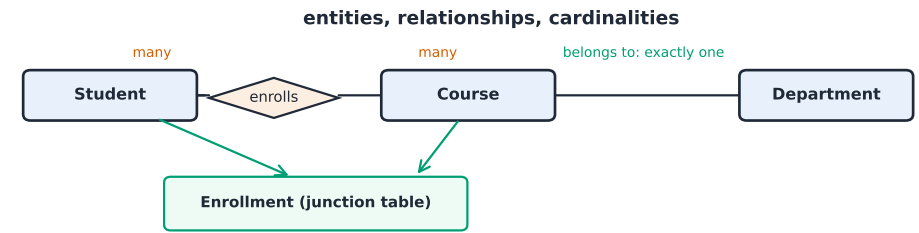
- Units 8.1–8.3 model and shape the data: ER, relational algebra and SQL, normalisation.
- Units 8.4–8.5 make it fast and safe: indexes with query plans, transactions with concurrency.
- Unit 8.6 governs it: security and data governance.

The database habit: **every claim about data is a query you can run** – each unit’s numbers come from the documented sample schema.

Relational modelling

Why it matters. The model decides everything downstream: what can be asked, what must be checked, what can go wrong. A good schema is a set of true sentences about the domain.

The idea, with a picture. An **entity** is a thing we count (a Student); a **relationship** connects entities (a Student **enrolls** in a Course). The **cardinality** of each end – one, many – becomes keys and tables: a many-to-many relationship becomes its own table of foreign-key pairs.



many-to-many becomes its own table of foreign-key pairs; PRIMARY KEY (student, course, term) enrolls once per term

referential integrity: every Enrollment.student and .course names a real row

sample data: 8 students - 6 courses - 14 enrollments

Students enroll in Courses (many-to-many through Enrollments); each Course belongs to one Department

The derivation (why keys exist). A **primary key** identifies a row uniquely: `student_id` in Student. A **foreign key** references it: `Enrollments.student_id` must name a real student – the database enforces **referential integrity**, so no enrollment can orphan itself. Our teaching schema:

```
Department(dept_id PK, name, building)
Student(student_id PK, name, dept_id FK)
Course(course_id PK, title, dept_id FK, credits)
Enrollment(student_id FK, course_id FK, term, grade,
            PRIMARY KEY (student_id, course_id, term))
```

Worked example 1 – modelling a rule. A student may enroll in a course at most once per term. The rule lives in the keys: Enrollments carries **PRIMARY KEY** (`student_id`, `course_id`, `term`) – a second insert of the same triple is rejected by the engine, not by the application. Rules in the schema are rules the database keeps.

Worked example 2 – the join path. Which students take a department’s courses? Student joins Enrollment on `student_id`, Enrollment joins Course on `course_id`, Course joins Department on `dept_id`: three joins, each following a foreign key. The suite’s sample data: 8 students, 6 courses, 14 enrollments – the join path yields 14 rows before filtering.

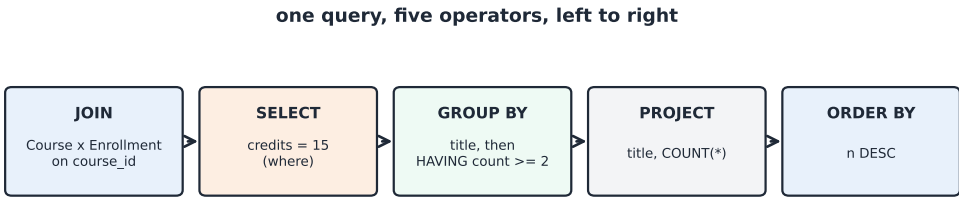
Common mistakes. Storing a student’s name inside Enrollment (duplicates, updates in many places). One table per relationship end instead of a junction table for many-to-many. Choosing a key that can change (emails move; surrogate IDs do not).

Summary. Entities to tables, relationships to foreign keys or junction tables; cardinality decides which; keys carry the rules the engine then enforces.

Relational algebra and SQL

Why it matters. SQL is the payroll language of the industry, and its heart is five algebra operators with exact semantics – know the algebra and every query is readable as a pipeline.

The idea, with a picture. The algebra: **selection** σ keeps rows ($\sigma_{credits=15}$), **projection** π keeps columns (π_{name}), **join** $\bowtie$ combines on a condition, union $\cup$ and set-difference – combine tables. Every SELECT is these five in a pipeline.



clause order is semantic: FROM builds rows, WHERE filters rows, GROUP BY partitions, HAVING filters groups

pipeline of the course query:

```
SELECT c.title, COUNT(*) AS n FROM Course c JOIN Enrollment e ON c.course_id = e.course_id
WHERE c.credits = 15 GROUP BY c.title HAVING COUNT(*) >= 2 ORDER BY n DESC;
```

executed result: Databases | 3

One query as a pipeline: from Courses select credits = 15, project the title, join Enrollments, project and count the student names

The derivation (WHERE before GROUP BY before HAVING). SQL’s clauses run in a fixed order: FROM builds the row set, WHERE filters rows, GROUP BY partitions them, HAVING filters groups, SELECT projects, ORDER BY sorts. So WHERE grade = 'A' keeps rows then groups; HAVING COUNT(*) >= 2 keeps groups only – filtering on an aggregate in WHERE is a category error the engine rejects.

Worked example 1 – one query, read as algebra.

```
SELECT c.title, COUNT(*) AS n
FROM Course c JOIN Enrollment e ON c.course_id = e.course_id
WHERE c.credits = 15
GROUP BY c.title
HAVING COUNT(*) >= 2
ORDER BY n DESC;
```

Algebra: join Course and Enrollment, select credits = 15, group by title, keep groups of size ≥ 2 , project title with the count. On the sample data the result is Databases | 3 – one 15-credit course with three enrollments (the suite records the exact output).

Worked example 2 – the outer join’s kept blanks. List every course with its enrollments: a plain join drops the 2 courses with no enrollment; LEFT JOIN keeps them with NULL counts – 6 rows, 2 with n = 0 (SQLite’s COUNT(e.student_id) counts present values only). The join flavor is a semantics decision, not a style one.

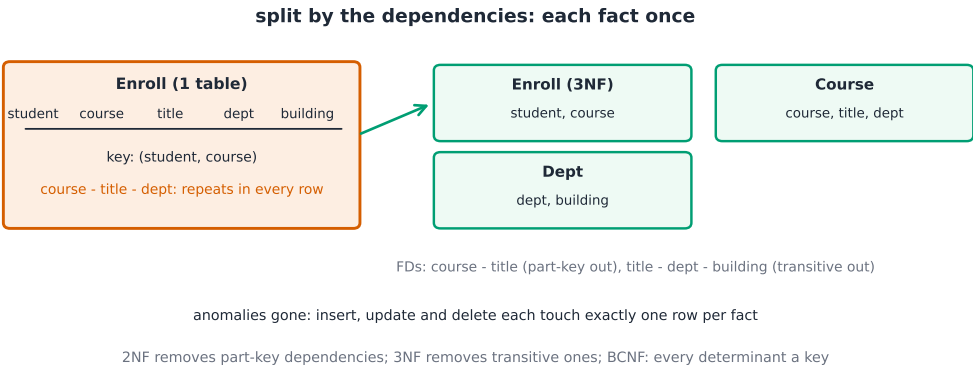
Common mistakes. Filtering aggregates in WHERE (use HAVING). Counting rows where you meant values (COUNT(*) vs COUNT(col) – the latter skips NULLs). Forgetting DISTINCT when projecting a many-side column (14 enrollment rows project 14 student ids, some repeated).

Summary. Five operators, one pipeline; clause order is semantic (WHERE rows, HAVING groups); join flavor decides who survives the match.

Normalisation

Why it matters. A denormalised table invites three anomalies: **insert**, **update**, **delete** – normalisation removes them by splitting along dependencies.

The idea, with a picture. A **functional dependency** $A \rightarrow B$: each A-value determines one B-value. **Second normal form** requires first normal form and no non-key attribute depending on only part of a candidate key. In **third normal form**, every nontrivial dependency has a superkey determinant or a right-hand attribute belonging to a candidate key. **Boyce–Codd BC** requires a superkey determinant for every nontrivial dependency.



One denormalised table split by its dependencies: the key part, the transitive department facts, and the enrollment fact itself

The derivation (why 3NF kills the anomalies). The un-normalised Enroll(student, course, title, dept, building) with key (student, course) holds: course $\rightarrow$ title (part-key), title $\rightarrow$ dept $\rightarrow$ building (transitive). Update anomaly: a course’s title appears in every enrollment row – change one, miss

ten. Split by the dependencies: `Course(course, title, dept)` and `Dept(dept, building)` and `Enroll(student, course)`: this removes the partial-key dependency from `Enroll`. Because `title → dept` still holds, `Course` is not yet in 3NF. Split it further into `Course(course, title)` and `Title(title, dept)`. With the stated dependencies, the four resulting tables are in BCNF.

Worked example 1 – grading the form. `Room(building, room, capacity, projector_type)` with key (building, room): if all rooms in a building share one projector type, the FD `building → projector_type` violates 2NF and therefore 3NF: a non-key attribute depends on only part of the candidate key. Split into `Building(building, projector_type)` and `Room(building, room, capacity)`. This removes the repeated projector fact.

Worked example 2 – when to stop. A warehouse report table stores `order_line` plus a running total per order: the total is derivable, but recomputing per view costs more than the anomaly risk of storing it – controlled denormalisation with a refresh trigger is a legitimate engineering choice once the normal form is understood. Normalise to reason; denormalise on evidence.

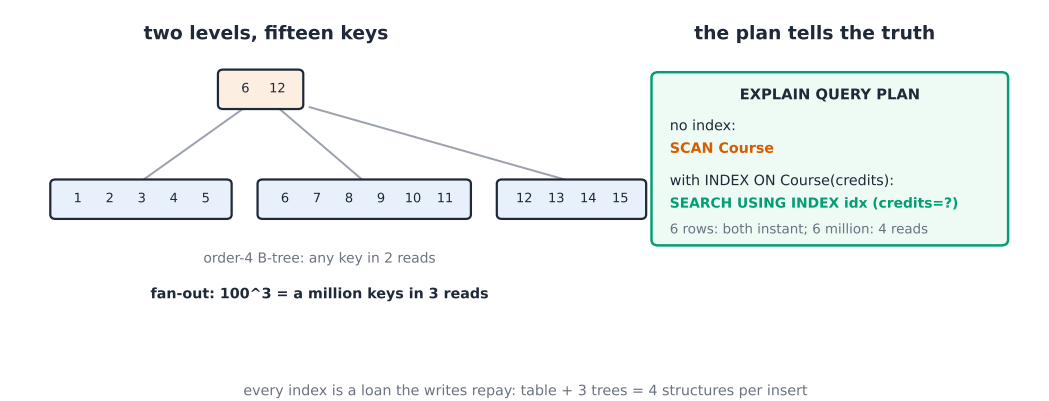
Common mistakes. Calling a table 3NF while a non-key column still determines another. Splitting so finely that every query joins five tables without measuring the cost. Forgetting the anomalies are symptoms – the disease is the repeated fact.

Summary. FDs drive the split: part-key out for 2NF, transitive out for 3NF in simple single-key examples; for BCNF, every nontrivial dependency has a superkey determinant. State the dependencies before judging a decomposition.

Indexes and query plans

Why it matters. A query that scans a million rows to find ten is a design failure, and the fix is not more hardware – it is an index the planner can use.

The idea, with a picture. A **B-tree** B index keeps keys sorted in a balanced multiway tree: our teaching tree (order 4, 15 keys) finds any key in $\lceil \log_4 15 \rceil = 2$ node reads against 15 sequential. The **query plan** is the engine’s chosen pipeline: scan or index, join order, strategy.



Left: the course-id B-tree, two levels for 15 keys. Right: the plan that uses it – index lookup instead of a full scan

The derivation (the fan-out arithmetic). A node holding k keys routes among $k + 1$ children: order m gives fan-out $\approx m$. Height h covers m^h entries: fan-out 100, three levels cover $100^3 = 10^6$ keys – a million-row lookup in 3 node reads. Every insert keeps the tree balanced (splits propagate upward), so the bound holds for every key, not just lucky ones.

Worked example 1 – a plan, read. `EXPLAIN QUERY PLAN SELECT title FROM Course WHERE credits = 20`: without an index the plan reads `SCAN Course`; with `CREATE INDEX idx ON Course(credits)` it reads `SEARCH Course USING INDEX idx (credits=?)`. On 6 rows both are instant; on 6 million the scan reads 6 million rows and the search reads 4 (fan-out 100 covers 10^6 keys at height 3 – six million needs height 4) – the plan, not the query text, prices the work.

Worked example 2 – the write tax. Each index adds work to every insert and delete: our schema with 3 indexes writes 4 structures per insert (table + 3 trees). A log table with 5 indexes and 10 million inserts per day maintains 50 million index entries daily – index what you query, drop what you do not, and measure.

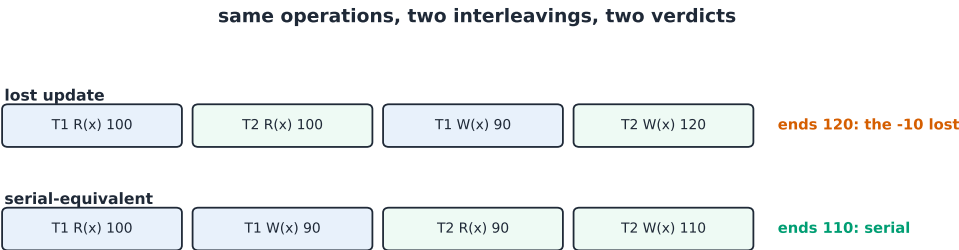
Common mistakes. Indexing every column (writes pay for reads you never make). Expecting the planner to use an index on a computed column (`WHERE UPPER(name) = ...` defeats a plain index). Reading the query instead of the plan.

Summary. B-trees trade fan-out for height: m^h keys in h reads; the plan tells the truth; every index is a loan the writes repay.

Transactions and concurrency

Why it matters. Two clerks, one balance, simultaneous updates: without transactions the database can commit both halves of different transfers and lose money invisibly.

The idea, with a picture. A transaction is all-or-nothing: **ACID** – atomicity, consistency, isolation, durability. Under **serializability** an interleaving’s result equals **SOME** serial order; weaker **isolation levels** trade correctness for concurrency: read uncommitted, read committed, repeatable read, serializable.



2PL: reads and writes lock, locks held to commit - one transaction waits, the outcome equals a serial order

isolation levels name the anomalies you accept: dirty read gone at read-committed, phantoms gone only at serializable

ACID: atomic - consistent - isolated - durable

Two transfers interleaved: locking reads and writes yields a serial-equivalent order; reading before the other’s commit yields a lost update

The derivation (the lost update, again). T1 reads balance 100, T2 reads 100, T1 writes 90, T2 writes 120: the 10-unit withdrawal has vanished – the banking twin of unit 7.3’s race. Two-phase locking (2PL) first acquires locks, then releases them; after the first release, no new lock may be acquired. This guarantees conflict serializability for transactions following the protocol. Strong strict 2PL additionally holds all locks until commit or abort. Transactions can deadlock; a detector may abort one so it can retry.

Worked example 1 – schedule legality. Interleaving R1(A) R2(A) W2(A) W1(A): T1’s write clobbers T2’s – a lost update, not serializable. Interleaving R1(A) W1(A) R2(A) W2(A): equals serial T1-then-T2 – legal. The suite checks both schedules by simulation.

Worked example 2 – the dirty read. At read-uncommitted, T2 reads T1’s uncommitted 90; T1 then rolls back – T2 has acted on a value that never existed. Read-committed prevents it (reads see only committed data); repeatable-read adds stability of re-reads; serializable adds phantoms’ absence. Each level is a promise about which anomalies remain.

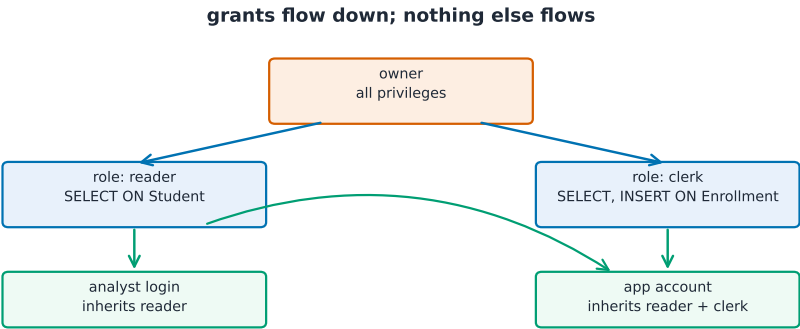
Common mistakes. Long transactions holding locks through user input (deadlock bait). Equating isolation with consistency (C is the constraints; I is the interleaving). Skipping the retry loop because deadlock “rarely” happens (it happens at peak).

Summary. ACID: all-or-nothing, constraints kept, interleaving equivalent to a serial order, commits that survive the crash; isolation levels name the anomalies you accept; 2PL buys serializability at the price of waiting.

Security and data governance

Why it matters. A database holds what people cannot afford to leak: the last decade’s breaches were access-model failures, not cryptographic ones.

The idea, with a picture. **Authentication** proves who you are; **authorisation** proves what you may do; the **principle of least privilege** shapes both: accounts get the narrowest grants that let the job run.



least privilege: the app account reads two tables and writes one - a stolen session is bounded by the grant

Grants flow from the owner to roles to users; the application account reads two tables and writes one – nothing else

The derivation (injection, and its end). A query built by pasting user text: "... WHERE name = ' + input + '" with input ' OR '1'='1 matches every row – and worse inputs drop tables. Parameterised statements bind values separately from SQL syntax, so text in a bound value cannot change the query structure. Table names, column names and other dynamic SQL need separate controls; parameter binding does not protect SQL still built by concatenation.

Worked example 1 – grants that fit. The teaching grants: GRANT SELECT ON Student TO reader; GRANT SELECT, INSERT ON Enrollment TO clerk; – the clerk account cannot read salaries it does not see, cannot DELETE an enrollment, and a stolen clerk session is bounded by the grant. Revoke on role change, not on audit day.

Worked example 2 – governance is a loop. Data governance

classifies data (public, internal, sensitive), assigns owners, sets retention, and audits access: the sensitive **grade** column is encrypted at rest, its access logged, and a quarterly review revokes the stale grants. The model survives contact with reality only because the loop runs.

Common mistakes. Connecting the application as the owner account (every bug runs with full authority). Auditing reads but not grants (privilege creep). Treating encryption as access control (it protects at rest, not against a valid login).

Summary. Authenticate, then authorise by least privilege; parameterise every query; classify, own, retain, audit – governance is the loop that keeps the model true.