

Topic 8: Databases

Transactions, indexes, richer SQL and a first look at NoSQL.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 8.1 Database concepts, 8.2 DBMS, 8.3 DDL and DML. Try the challenge questions on the last pages before you read the answers.

Transactions and ACID

A bank transfer needs two updates: take £100 from account 1 and add £100 to account 2. If the system fails between them, £100 vanishes. A **transaction** groups statements so they succeed or fail together:

```
BEGIN TRANSACTION;
UPDATE Account SET Balance = Balance - 100 WHERE ID = 1;
UPDATE Account SET Balance = Balance + 100 WHERE ID = 2;
COMMIT; -- or ROLLBACK; to undo both
```

Property	Meaning
Atomicity	All of the transaction happens or none of it does.
Consistency	The database moves from one valid state to another; rules and constraints still hold.
Isolation	Concurrent transactions do not see each other's half-finished work (prevents lost updates).
Durability	Once committed, the change survives a crash or power cut (via a write-ahead log).

Indexes: how a database finds a row fast

Without help, finding `WHERE Surname = 'Patel'` means scanning every row: 1 000 000 rows, up to 1 000 000 checks. An **index** is a separate sorted structure (usually a **B-tree**) that maps key values to row locations. A B-tree where each node holds ~100 keys is only about three levels deep for a million rows, so a lookup reads ~3 nodes.

```
CREATE INDEX idx_surname ON Student(Surname);
```

The trade-off: an index uses extra storage and every INSERT, UPDATE or DELETE must also update it. Index the columns you search, join and sort by often, not everything. Primary keys are indexed automatically.

Beyond basic SELECT

A **LEFT JOIN** keeps every row from the left table even when there is no match, filling gaps with NULL. Combined with **GROUP BY** and **HAVING** it answers questions like "which students have never borrowed a book?":

```
SELECT Student.Name, COUNT(Loan.ID) AS Loans
FROM Student LEFT JOIN Loan ON Student.ID = Loan.StudentID
GROUP BY Student.ID, Student.Name
HAVING COUNT(Loan.ID) = 0;
```

WHERE filters individual rows *before* grouping; **HAVING** filters whole groups *after* aggregation. A **subquery** puts one query inside another, e.g. `WHERE Price > (SELECT AVG(Price) FROM Product)`.

NoSQL: when tables are the wrong shape

Relational databases excel at structured, related data with strong guarantees. **NoSQL** systems trade some of that for flexibility and scale: **document stores** (JSON documents with varying fields), **key-value stores**, **graph databases** (for relationships such as social networks) and **column stores**. They often scale across many servers more easily but may offer only *eventual* consistency and weaker joins. The right choice depends on the data and the guarantees needed.

Challenge questions

1. Power fails after the first UPDATE of a bank transfer but before the second. When the database restarts, account 1 has lost £100 but account 2 gained nothing. Which ACID property has been violated, and how should a DBMS prevent it?
2. A 500 000-row Customer table is searched by Surname thousands of times a day. Suggest an improvement and one drawback.
3. Tables: Book(ID, Title, Genre) and Loan(ID, BookID, StudentID). Write SQL listing each genre with more than 100 loans.

Worked answers

Only look once you have had a proper go at each question.

1. **Atomicity.** The two updates form one transaction; the DBMS should log changes and, on restart, roll back any uncommitted transaction so neither update takes effect.
2. Create an index on Surname (`CREATE INDEX idx_surname ON Customer(Surname);`) so searches use a B-tree lookup instead of scanning every row. Drawback: extra storage, and slower INSERT/UPDATE/DELETE because the index must be maintained.
3. Join the tables, group by genre and keep only the groups over the limit:

```
SELECT Genre, COUNT(*)  
FROM Book JOIN Loan ON Book.ID = Loan.BookID  
GROUP BY Genre  
HAVING COUNT(*) > 100;
```

HAVING (not WHERE) is needed because the condition applies to the group count.