Cambridge Part IA

Databases

Complete Exam Revision Guide

Lectures 1–8 • Michaelmas 2025–26 • Paper 3

What's Inside

Covers every topic from the slides (djg11), all past Tripos questions 2020–2025, model answers, and exam technique advice.

1 Foundations & Terminology

1.1 What Is a DBMS?

A Database Management System (DBMS) provides a standardised, high-level interface (typically SQL) that sits between applications and physical storage. It hides low-level data layout, supports concurrent access, enforces consistency rules, and manages transactions.

The 'narrow waist' model: many possible applications above the interface, many possible storage implementations below — the interface decouples them.

Core Terminology

Term	Definition
Value	A piece of data — string, number, date, polygon, etc.
Field / Attribute / Column	A named slot that holds one value per record
Record / Row / Tuple	One complete entry; a sequence of fields
Schema	Specification of table name, field names, types, key, constraints
Key	Field(s) whose value uniquely identifies a record in a table
Primary Key	The chosen key used to identify records (must be unique, not NULL)
Foreign Key	A field in one table whose value must appear as a key in another table
Index	A derived data structure enabling fast record lookup (log n vs. n cost)
Query	A retrieval operation, often with automated query planning
Transaction	An atomic unit of work with ACID properties
CRUD	Create, Read, Update, Delete — the four basic database operations

Three Data Models in This Course

Model	Characteristics	System Used
Relational (SQL)	Data in 2D tables. High update throughput. Industry standard for 48+ years.	SQLite
Document-oriented	Semi-structured JSON/XML documents. Optimised for	TinyDB

	reads, few updates.	
Graph-oriented	Nodes and edges. Path queries, graph algorithms. Efficient traversal.	Neo4j (Cypher)

Key principle: No single model is ideal for all problems — every design involves trade-offs.

2 Entity-Relationship (ER) Diagrams

ER diagrams are an implementation-independent way to model data, due to Peter Chen (1976). They describe the conceptual structure before deciding on any database technology.

2.1 Core Components

Symbol	Meaning
Rectangle (square)	Entity — represents a real-world 'noun' (e.g. Movie, Person)
Oval / Ellipse	Attribute — a property of an entity or relationship (e.g. title, name)
Diamond	Relationship — a verb connecting entities (e.g. Directed, Acted_In)
Underlined attribute	Key attribute — uniquely identifies an entity instance
Arrow (→) on relationship	Many-to-one: the arrowhead end is the 'one' side
No arrow (both sides open)	Many-to-many relationship
Two arrows (← →)	One-to-one relationship (rare in practice)
Double rectangle	Weak entity — existence depends on another entity
Dashed oval	Discriminator — partial key for weak entities
IsA triangle	Entity hierarchy / inheritance (sub-entities inherit parent's attributes)

2.2 Relationship Cardinalities

Cardinality tells us how many instances of one entity relate to how many instances of another.

Cardinality	Explanation
Many-to-many (M:N)	Any S can relate to 0+ T's; any T can relate to 0+ S's. No arrows. E.g. Movie ↔ Person via Directed.
One-to-many (1:M)	Each S relates to 0+ T's but each T relates to at most one S. Arrow on S-side. E.g. Department ← Employee.
Many-to-one (M:1)	Same as 1:M viewed from the other direction. Arrow on T-side.
One-to-one (1:1)	Each S relates to at most one T and vice

	versa. Arrows on both sides. Very rare — ask 'why not merge?'

2.3 Weak Entities

A weak entity cannot exist without its 'owner' entity. Its discriminator (dashed oval) is a partial key — combined with the owner's key it uniquely identifies the weak entity.

Example: AlternativeTitle is a weak entity of Movie. To find a specific alternative title you need both movie_id and alt_id (the discriminator).

2.4 Entity Hierarchies (IsA)

An IsA relationship models specialisation. Sub-entities inherit all attributes and relationships from the parent entity. E.g. Temporary_Employee and Contract_Employee both IsA Employee.

In implementation there are three main strategies (see Section 3.6):

1. Three separate tables (parent + two sub-tables sharing the parent key)
2. Two tables (one merged parent+sub, one for the other sub)
3. One table with type-tag columns (NULLs used for inapplicable attributes)

2.5 Modelling Pitfalls to Know

- Attributes exist at most once per entity/relation — if you need multiple values (e.g. multiple roles), you need a separate entity or relationship table.
- Multi-valued attributes violate First Normal Form (1NF / value atomicity) when stored in a real database as comma-separated lists.
- Ternary vs. binary relationships: a ternary can sometimes be replaced with three binary relationships via an intermediary entity, but this can lose information (see textbook 3.2.6).
- Attribute vs. entity: if the 'attribute' has multiple values per record, or has its own properties you need to query, it should become an entity with its own table.

3 The Relational Model

Codd's radical idea (1970s): give users a model and query language completely independent of physical data representation. Based on mathematical relations (set theory).

3.1 Mathematical Foundations

A relation R over domains S_1, S_2, …, S_n is any finite set $R \subseteq S_1 \times S_2 \times \ldots \times S_n$. In a database, columns are named (attributes), column order does not matter, and row order usually does not matter.

Schema notation: $R(A_1: S_1, A_2: S_2, \ldots, A_n: S_n)$. Simplified: R(A, B, C) where underline indicates key.

3.2 Keys & Foreign Keys (Formal Definitions)

Formal Definitions

Superkey Z: For any records u,v in R, $u.[Z] = v.[Z] \implies u.[X] = v.[X]$ (Z uniquely determines every field)

Key Z: A minimal superkey — no proper subset of Z is also a superkey.

Foreign key: Z in S is a foreign key for R when $\pi Z(S) \subseteq \pi Z(R)$ (every Z-value in S must appear in R).

Referential integrity: all foreign key constraints are satisfied in the current database state.

All-key table: Every attribute is part of the key (common for relationship tables like Has_Genre, plays_role, has_position).

3.3 Implementing ER in Relational Tables

Both ER entities AND ER relationships are implemented as tables. Common patterns:

Scenario	Implementation Strategy
M:N relationship	New all-key table R(key_S, key_T, attrs) with two foreign keys
1:M relationship (clean)	New table R(key_S, key_T, attrs) — or expand T to include a nullable FK
1:1 relationship	Expand one of the entity tables with a nullable FK column for the other
Weak entity	Table T(parent_key, discriminator, attrs) with FK on parent_key
Multiple relationships	Can share one table with a type-tag column and NULL for inapplicable attrs
IsA hierarchy (3-table)	Parent table S(Z, W); sub-tables T(Z, Y), U(Z, V) each with FK to S

3.4 Problems with Data Redundancy

Data is redundant if it can be deleted and reconstructed from remaining data. Redundancy causes three update anomalies:

Anomaly Type	Example
Insertion anomaly	Cannot insert entity without knowing all related data (e.g. a person without a film)
Deletion anomaly	Deleting all films by a director loses all information about that director
Update anomaly	A director's misspelled name updated in one row but not others — inconsistency

Solution: break tables down so each table has one clear purpose. Each piece of data appears in exactly one place ('one fact, one place').

4 Relational Algebra (RA)

The RA provides a clean mathematical foundation for understanding SQL. Queries take one or more relation instances as input and produce one relation instance as output.

4.1 The Six Core Operators

Operator	Meaning	SQL Equivalent
$\sigma_p(R)$ Selection	Returns rows of R satisfying predicate p	SELECT * FROM R WHERE p
$\pi_x(R)$ Projection	Returns only columns X from R (set semantics: dedup)	SELECT DISTINCT X FROM R
$\rho_m(R)$ Renaming	Renames attributes per mapping M	SELECT A AS E FROM R
$R \cup S$ Union	All rows in R or S (schemas must match)	(SELECT * FROM R) UNION (SELECT * FROM S)
$R - S$ Difference	Rows in R but not in S	(SELECT * FROM R) EXCEPT (SELECT * FROM S)
$R \times S$ Product	All combinations of rows from R and S (cross join)	SELECT * FROM R CROSS JOIN S
$R \cap S$ Intersection	Rows appearing in both R and S	(SELECT * FROM R) INTERSECT (SELECT * FROM S)
$R \bowtie S$ Natural Join	Join on all common attribute names, project out duplicates	SELECT * FROM R NATURAL JOIN S

4.2 Natural Join — Formal Definition

Natural Join

Given R(A, B) and S(B, C):

$R \bowtie S = \{ t \mid \exists u \in R, v \in S,\ u.[B] = v.[B] \ \wedge \ t = u.[A] \cup u.[B] \cup v.[C] \}$

In RA: $R \bowtie S = \pi_{A,B,C}(\sigma_{B=B'}(R \times \rho_{\{B \to B'\}}(S)))$

Join is associative: $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

This matters for query optimisation — different join orderings have very different costs.

4.3 Union / Intersection / Difference Requirements

Schema Compatibility Rules

Union ($R \cup S$): Both R and S must have the SAME set of attribute names (same

schema).
  If they don't share names, use renaming (ρ) first.
Intersection (R ∩ S): Same schema requirement.
Difference (R − S): Same schema requirement.
Product (R × S): R and S must have DISJOINT attribute names.
  If names clash, use ρ to rename before taking the product.

Exam tip (2024 Q1a): Union DOES require schemas to share attribute names.
The same consideration applies to intersection (same answer).

4.4 Sizes After Operations

If R has P records and S has Q records:

Operation	Resulting Size Range
R ∪ S	Min: max(P, Q) — if one is a subset of the other; Max: P + Q — if no overlap
R ∩ S	Min: 0 — if disjoint; Max: min(P, Q) — if one is a subset
R − S	Min: 0 — if R ⊆ S; Max: P — if S and R are disjoint
R × S	Always exactly P × Q
R ⋈ S	Min: 0 — if no matching values; Max: P × Q — if all B-values match

5 SQL — The Essentials

5.1 SELECT Anatomy

SQL SELECT Template

```
SELECT [DISTINCT] col1, col2, ...     -- what columns to return
FROM   table1 [AS alias1]             -- which tables
[JOIN  table2 AS alias2               -- join another table
  ON   alias2.fk = alias1.pk]         -- join condition
[WHERE condition]                     -- filter rows
[GROUP BY col1, ...]                  -- group into buckets
[HAVING aggregate_condition]          -- filter groups
[ORDER BY col [ASC|DESC]]             -- sort result
[LIMIT n]                             -- truncate result
```

5.2 SELECT vs SELECT DISTINCT

Key Distinction — Bags vs. Sets

SQL is based on MULTISETS (bags), not sets.

SELECT returns a multiset — duplicate rows CAN appear if projecting onto fewer columns.
SELECT DISTINCT returns a set — duplicates are removed.

This is crucial for the 2020 Q1 questions (parts c and d):

- Without DISTINCT: SELECT sid, uid can return more rows than SELECT A, C if multiple (sid, uid) pairs share the same (A, C) values.
- With DISTINCT: SELECT DISTINCT A, C can return more rows than SELECT DISTINCT sid, uid
  because different (sid, uid) pairs might map to the same (A, C) values.
  Wait — actually SELECT DISTINCT A, C ≤ SELECT DISTINCT sid, uid in record count
  because each unique (sid, uid) pair has unique (A, C) only if A and C are injective.
  In general, DISTINCT sid,uid ≥ DISTINCT A,C is not guaranteed either way.

5.3 JOINs

Join Type	Behaviour
CROSS JOIN (or comma in FROM)	Cartesian product — all row combinations
JOIN … ON condition (INNER JOIN)	Only rows satisfying the join condition
NATURAL JOIN	Auto-join on all shared column names, dedup

	shared cols
LEFT OUTER JOIN	All rows from left table; NULL filled for non-matching right rows
RIGHT OUTER JOIN	All rows from right table; NULL filled for non-matching left rows
FULL OUTER JOIN	All rows from both; NULLs where no match
Self-join (table AS a JOIN table AS b)	Join a table with itself — essential for path/co-actor queries

Complexity: Brute force join is O(|R| × |S|). With an index on S.B the effective cost is O(|R| × log|S|).

5.4 GROUP BY and Aggregate Functions

GROUP BY partitions rows into groups with equal values in the specified columns. Each group then collapses to one row via an aggregate function.

Why aggregation is required: After GROUP BY, each group contains multiple rows. You cannot return a non-group column directly — only group keys or aggregate results. Returning a raw column would be ambiguous (which row's value?).

Aggregate Function	Meaning
COUNT(*)	Number of rows in the group (including NULLs)
COUNT(col)	Number of non-NULL values of col
SUM(col)	Sum of non-NULL values
AVG(col)	Arithmetic mean of non-NULL values
MIN(col) / MAX(col)	Minimum / maximum non-NULL value

Mathematical requirement for aggregates: The function must produce a single value from a multiset of values. It should be insensitive to the order of its inputs (commutative, associative) to allow any evaluation order — this is key for query optimisation and parallel execution.

5.5 Subqueries and NOT IN

Anti-join Patterns

```
-- Anti-join: rows in R not related to anything in S
SELECT sid FROM S
WHERE sid NOT IN (SELECT sid FROM R);

-- Equivalent with LEFT JOIN / IS NULL pattern:
SELECT S.sid FROM S
LEFT JOIN R ON R.sid = S.sid
WHERE R.sid IS NULL;
```

WARNING: NOT IN with NULLs can give unexpected results (see NULL section).

5.6 Schema DDL (CREATE TABLE)

DDL Example

```
CREATE TABLE genres (
  genre_id  INTEGER  NOT NULL,
  genre     TEXT     NOT NULL,
  PRIMARY KEY (genre_id)
);

-- All-key table with two foreign keys:
CREATE TABLE has_genre (
  movie_id  VARCHAR(16)  NOT NULL  REFERENCES movies(movie_id),
  genre_id  INTEGER      NOT NULL  REFERENCES genres(genre_id),
  PRIMARY KEY (movie_id, genre_id)
);
```

5.7 NULL Values & Three-Valued Logic

NULL — Critical Rules

NULL is NOT a value — it is a placeholder meaning 'unknown/inapplicable'.
NULL is not the empty string ''.
NULL is not equal to anything, not even another NULL.
(NULL = NULL) evaluates to NULL (not TRUE).

Three-valued logic truth tables:
AND: T∧T=T, T∧F=F, T∧⊥=⊥, F∧F=F, F∧⊥=F, ⊥∧⊥=⊥
OR: T∨T=T, T∨F=T, T∨⊥=T, F∨F=F, F∨⊥=⊥, ⊥∨⊥=⊥
NOT: ¬T=F, ¬F=T, ¬⊥=⊥

To test for NULL use: col IS NULL or col IS NOT NULL
(NOT col = NULL is always NULL, never TRUE/FALSE)

WHERE clause: only rows where condition evaluates to TRUE are returned.
Rows where condition is NULL (unknown) are EXCLUDED — this catches people out!
e.g. WHERE age <> 19 will NOT return rows where age IS NULL.

Inconsistency in GROUP BY: NULL = NULL is FALSE for equality, but GROUP BY
treats all NULLs as belonging to the same group (SQL implementation anomaly).

5.8 Recursive SQL (WITH RECURSIVE)

The RA cannot compute transitive closure (because we don't know k in advance). SQL extends RA with WITH RECURSIVE to handle this.

Recursive Bacon Numbers

```
WITH RECURSIVE bacon(n, pid) AS (
  -- Base case: Kevin Bacon himself
  SELECT 0 AS n, pid2 AS pid FROM coactors
    WHERE pid1='nm0000102' AND pid1=pid2
  UNION
  -- Recursive step: extend one hop
  SELECT n+1, c.pid2 FROM bacon
  JOIN coactors AS c ON c.pid1 = pid
  WHERE NOT(c.pid2 IN (SELECT pid FROM bacon)) AND n < 20
)
SELECT n, COUNT(*) FROM (SELECT min(n) AS n, pid FROM bacon GROUP BY pid)
GROUP BY n;
```

Transitive closure $R^+ = \cup\{R^n : n \geq 1\}$ cannot be computed in plain RA because k (the depth) depends on the data and is not known at query-compile time.

6 Transactions, ACID, Consistency & Throughput

6.1 ACID Properties

Property	Meaning
Atomicity	All changes happen or none do. A transaction either commits fully or is completely rolled back.
Consistency	Every transaction takes the database from one consistent state to another. User-defined invariants are preserved.
Isolation	Concurrent transactions appear to execute serially. Intermediate states are invisible to others.
Durability	Once committed, changes persist even across system failures (written to non-volatile storage).

6.2 BASE (Eventual Consistency)

Property	Meaning
BA — Basically Available	Availability is prioritised over strict consistency. Some replica always responds.
S — Soft State	The system state may change without explicit application action (due to ongoing eventual-consistency updates).
E — Eventual Consistency	If update activity stops, all replicas will eventually converge to the same state.

Advantage: Higher availability and write throughput, especially in distributed systems.

Disadvantage: No guarantee of immediately consistent reads. Stale/conflicting data visible temporarily. Not suitable for scenarios requiring strong consistency (e.g. bank account transfers).

6.3 CAP Theorem

CAP Theorem — Cannot Have All Three

In any distributed database, you can guarantee at most TWO of the following three:

C — Consistency: All reads return up-to-date data
A — Availability: All clients can find some replica of the data
P — Partition Tolerance: System operates despite message loss or partial failure

It is impossible to achieve all three simultaneously with current (pre-quantum) technology.

Exam tip (2024 Q2a): Know these definitions precisely and explain WHY.
Reason: During a network partition, if you require both consistency and availability, you must respond even when you cannot verify data is up-to-date — contradiction.

6.4 Redundancy, Throughput & Normalisation

Strategy	Effect
Low redundancy	Good for high UPDATE throughput (fewer locks needed). Risk of stale read caches.
High redundancy	Good for fast READS (pre-computed answers, fewer joins). Slow updates (many copies to keep consistent).
Indexes	Speed up reads (log n lookup) but slow down writes (index must be updated). Classic read-write tradeoff.
Normalisation	Reducing redundancy by splitting tables. Good for update-heavy (OLTP) workloads.
Denormalisation	Intentional redundancy for read-heavy (OLAP/document) workloads.

6.5 OLAP vs OLTP

Characteristic	OLAP	OLTP
Purpose	Analysis, decision support, data warehousing	Day-to-day operations
Data type	Historical / archived	Current / live
Transactions	Mostly reads (large scans)	Mostly updates
Redundancy	High (denormalised, precomputed)	Low (normalised)
DB size	Humongous	Large
Consistency	Eventual / periodic snapshots (ETL)	ACID required

ETL = Extract, Transform, Load — process for moving data from OLTP to OLAP warehouse.

7 Normalisation & Functional Dependencies

A normalised database is essentially one with little or no redundant data.

7.1 The Core Rule

Normalisation Principle
All data in a table should either:
(a) be part of the key, OR
(b) semantically (functionally) depend on the whole key.

If some data depends only on part of the key, or on a non-key attribute,
that data should be split into a separate table.

Example of violation: Table with (movie_id, person_id, director_name, movie_title)
director_name depends on person_id alone (not the full key)
movie_title depends on movie_id alone (not the full key)
Split: movies(movie_id, movie_title), people(person_id, director_name),
directed(movie_id, person_id)

Note: The subtleties of 3NF vs BCNF are NOT on the Part IA syllabus. Just know the principle of functional dependence on the key.

7.2 Functional Dependencies — Spot the Violation

Spotting redundancy: If a value of field F is always predictable from another field G (not the key), then there's a functional dependency G → F. This means F is redundant and should be split off into a table keyed by G.

Exam example (2024 Q2c): R3(A, B, D, E, F) where F is always predictable from E. This means E→F is a functional dependency. Split off R3a(E, F) and R3b(A, B, D, E). If updates are rare, the precomputed F may stay.

8 Document-Oriented Databases

8.1 Semi-Structured Data & JSON/XML

Semi-structured data has some structure, but not the rigid schema of a relational table. Parts are free-text; parts are tagged/named values.

Format	Structure
JSON (JavaScript Object Notation)	Tree-structured text. OBJECT (key/value dict), ARRAY (ordered list), LEAF_S (string), LEAF_N (number), NULL. Unordered keys.
XML (eXtensible Markup Language)	Tree with named ELEMENT nodes (with attributes as string pairs) and LEAF string nodes. Designed for markup and serialisation.

Both are broadly similar: tree-structured text with named nodes. Both are used for data serialisation (moving data between systems) and as the native storage format in NoSQL systems.

8.2 Document Databases vs. Relational

Aspect	Relational
Schema	Rigid, enforced by DBMS
Data shape	Flat 2D tables
Joins	Explicit JOIN required
Updates	Excellent (low redundancy)
Reads	Multiple joins needed
Consistency	ACID (typically)
Distribution	Hard to distribute correctly
Foreign key integrity	Enforced

8.3 Denormalisation Advantage

In a distributed document database, a round-trip to retrieve data is expensive. A denormalised document bundles all likely-needed data under one key so a single fetch retrieves everything.

Different key nestings: the same relational data may be stored multiple times, once with key A at the top and once with key B at the top, to support efficient lookup by either key. This multiplies storage but eliminates multi-hop lookups.

Misspelling cost: In a document DB, correcting a misspelled actor name requires updating every document where that name appears — potentially thousands of records (the update anomaly problem). In a relational DB, you fix one row in the people table.

8.4 JSON in SQL / Hybrid Approaches

Modern rDBMS systems support JSON fields with query capabilities. Two useful JSON expansion operations:

Operation	Description
JSON_TABLE / json_each (lateral flatten)	Expands a JSON array field into multiple rows — one row per array element. Makes each array element available as a relational row for SQL querying.
JSON_EXTRACT / ->> operator	Extracts a specific field from a JSON object by path. Returns a scalar value usable in WHERE clauses, SELECT, etc.

Problems that remain with JSON-in-SQL: Type safety is weakened (all JSON values are strings/numbers). Nested structure may need multiple expansion passes. Schema validation is limited. Performance for deep nested queries can be poor. Multiple JSON fields each need separate expansion.

9 Graph-Oriented Databases

9.1 Graph Structure

G = (V, A) where V is a finite set of nodes (vertices) and A ⊆ V × V is a set of directed edges (arcs).

Concept	Details
Nodes	Have a type, a unique label, and properties (key-value pairs). E.g. (nm0000102:Person {name: 'Kevin Bacon', birthyear: 1958})
Edges	Directed, typed, optionally labelled, with properties. E.g. (nm0002002)-[:ACTED_IN {Role:'James Bond'}]->(tt0299478)
Endorelation	A relation from a domain to itself (e.g. co-actors, city connections). Often symmetric.
Transitive closure	R^+ — all pairs reachable by any number of hops. Built into graph databases; requires recursion in SQL.

9.2 Why Graphs, Not Just Relational Tables?

Storing a graph in relational tables (NODES + EDGES tables) is possible but inefficient:

- All edges must be scanned to find neighbours of a node (no pointer-based traversal).
- Multi-hop queries require recursive SQL or many self-joins — painful in practice.
- Typically need two inverted indexes on the edges relation (one per endpoint).
- In-core graph DBs can use actual pointers between nodes — O(1) neighbour lookup.

9.3 Why One Graph?

A graph database typically holds just one big graph because its power comes from traversal across all node types and edge types simultaneously. Splitting into multiple graphs would require explicit joins at the application level, losing the path-query advantage. The single graph is essentially an instantiation of the whole ER diagram — all entities and all relationships live in the same graph and can be traversed freely.

9.4 Cypher Query Language (Neo4j)

Cypher Examples

```
-- Friends-of-friends:
MATCH (john {name:'John'})-[:FRIEND]->()-[:FRIEND]->(fof)
RETURN john.name, fof.name

-- Co-actors (two equivalent forms):
```

```
MATCH (p1:Person)-[:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(p2:Person)
WHERE p1.person_id <> p2.person_id
RETURN p1.name, p2.name, count(*)

-- Bacon numbers (all shortest paths):
MATCH path=allshortestpaths(
  (m:Person {name:'Kevin Bacon'})-[:ACTED_IN*]-(n:Person))
WHERE n.person_id <> m.person_id
RETURN length(path)/2 AS bacon_number, COUNT(distinct n) AS total
ORDER BY bacon_number
```

9.5 Relational Composition & Bacon Numbers

Relation composition: Given $R \subseteq S \times T$ and $Q \subseteq T \times U$, their composition $Q \circ R \subseteq S \times U = \{(s,u) \mid \exists t.(s,t) \in R \wedge (t,u) \in Q\}$. This is the mathematical foundation for the join operation.

Bacon number k: actor X has Bacon number k if the shortest path in the co-actor graph from Kevin Bacon to X has k edges. R = co-actor relation. Bacon number k iff (Kevin_Bacon, X) $\in R^k$ but not in R^n for any $n < k$.

10 Past Tripos Questions — Model Answers

2020 Paper 3 Q1 — Joins and Paths

Setup
Tables: S(sid, A), T(tid, B), U(uid, C), R(sid, tid, D), Q(tid, uid, E)
R = many-to-many between S and T, Q = many-to-many between T and U

(a) All (sid, uid) pairs related through R and Q:

```
SELECT R.sid, Q.uid
FROM   R
JOIN   Q ON Q.tid = R.tid;
```

Note: use SELECT not DISTINCT — the question asks for it without DISTINCT. Duplicates arise when multiple T records link the same (sid, uid) pair.

(b) All (A, C) pairs related through R and Q:

```
SELECT S.A, U.C
FROM   R
JOIN   Q ON Q.tid  = R.tid
JOIN   S ON S.sid  = R.sid
JOIN   U ON U.uid  = Q.uid;
```

(c) Can one return more records than the other?

Answer
YES — query (a) can return MORE records than query (b).

Reason: query (a) selects (sid, uid) which is more specific — multiple distinct (sid, uid) pairs may share the same (A, C) values if A is not injective on sid, or C on uid. Without DISTINCT, both return multisets. Query (a)'s multiset has one row per (sid, uid, tid) combination. Query (b) maps each such row to (A, C). If multiple sids have the same A-value, the number of (A,C) rows can be the same or differ.

More precisely: (a) returns one row per (sid, tid, uid) path. (b) collapses sid→A and uid→C (functions, since A is determined by sid and C by uid). So (b) returns at most as many rows as (a). It can return FEWER if multiple sids share an A value.

(d) With DISTINCT — can one return more?

Answer
With DISTINCT:
(a) returns DISTINCT (sid, uid) pairs

(b) returns DISTINCT (A, C) pairs

YES — query (b) DISTINCT A,C can return fewer records than (a) DISTINCT sid,uid. Multiple distinct (sid, uid) pairs can map to the same (A, C) pair (if two different sids happen to have the same A value, and two different uids have the same C value). So DISTINCT A,C ≤ DISTINCT sid,uid in record count.

Therefore (a) with DISTINCT can return MORE (or equal) records than (b) with DISTINCT.

(e) COUNT paths through each T record:

```
SELECT R.tid, COUNT(*) AS total
FROM   R
JOIN   Q ON Q.tid = R.tid
GROUP BY R.tid;
```

Explanation: For a given tid, each row in R with that tid represents an S-entry that can reach T, and each row in Q with that tid represents a U-entry reachable from T. The join R JOIN Q ON Q.tid=R.tid produces one row per complete path (sid→tid→uid). Grouping by tid and counting gives the total number of paths through each T record.

To include T records with no paths (0 paths), use LEFT JOINs:

```
SELECT T.tid, COUNT(R.sid) AS total   -- COUNT(R.sid) counts non-NULL matches
FROM   T
LEFT JOIN R ON R.tid = T.tid
LEFT JOIN Q ON Q.tid = T.tid
GROUP BY T.tid;
```

2020 Paper 3 Q2 — DISTINCT and Conclusions

Setup
Tables: S(sid, A), T(tid, B), R(sid, tid, C) — C is part of R's key
Q1 = SELECT S1.A, R1.C ... (no DISTINCT)
Q2 = SELECT DISTINCT S1.A, R1.C ... (same query, with DISTINCT)

(a) What does C being in R's key mean?

C is part of the primary key, meaning that (sid, tid, C) together uniquely identify a row. Multiple records with the same (sid, tid) pair can exist — they are distinguished by different C values. This means the same S-T pair can be related through multiple different 'C-labelled' relationships.

(b) sids of S-records not R-related to any T:

```
SELECT sid FROM S
WHERE  sid NOT IN (SELECT sid FROM R);
```

```
-- Alternative:
SELECT S.sid FROM S
LEFT JOIN R ON R.sid = S.sid
WHERE R.sid IS NULL;
```

(c)(i) If Q1 = Q2 (no WHERE clause):

Answer
Q1 can have more rows than Q2 due to duplicate (A, C) pairs in the multiset.
Q1 = Q2 means there are NO duplicate (A, C) pairs in the Q1 result.

Conclusion: Every combination of (S.A, R.C) produced by the join appears exactly once.
This means: for each A-value, the set of C-values it appears with has no repeats.
Equivalently: for any two R-records (r1, r2) with the same tid, the combination (S[r1.sid].A, r1.C) ≠ (S[r2.sid].A, r2.C) for all such pairs.

(c)(ii) With WHERE R1.C = R2.C AND S1.sid ≠ S2.sid:

Answer
This finds: pairs of R-records sharing the same tid AND same C value, but linked to different S-records.

Q1 = Q2 means: no (A, C) pair appears more than once in the result.

Conclusion: For any two distinct S-records s1, s2 that share a T-record via R with the same C value, their A values must be distinct (A1 ≠ A2).
In other words: no two different S-entries with the same (tid, C) combination have the same A value.

(c)(iii) With WHERE R1.tid ≠ R2.tid:

Answer
This finds: pairs of R-records sharing the same S-record (via S1.sid = R1.sid, S2.sid = R2.sid — note S1/S2 come from different R1/R2 joins) but with different T-records.

Wait — re-read the query. Both joins go through S on sid=R.sid. R1 and R2 both join to S independently. So we get all pairs (R1, R2) where R1.tid ≠ R2.tid.

Q1=Q2: no duplicate (S1.A, R1.C) pairs arise.
Conclusion: For each distinct A-value (from S), the set of C-values from R-records linked via different T-records is pairwise distinct. I.e. the same S's A-value does not appear with the same C-value from two different T-records.

(c)(iv) With WHERE R1.tid ≠ R2.tid AND S1.sid ≠ S2.sid:

Answer
Now we require: different S-records AND different T-records.
Q1=Q2: no duplicate (A, C) pair across all such path-pairs.

Conclusion: No two distinct S-entries (with different sids) produce the same (A, C) combination when each is linked (via possibly different T-records) to R-records with the same C value. I.e. A uniquely identifies the C values in this constrained join.

2021 Paper 3 Q1 — Library Database & ER Redesign

Original Schema
Book(book_id, title, number_owned, number_borrowed)
Person(person_id, name, address)
Borrowed(person_id, book_id, number) -- person_id, book_id are FKs

(a) Consistency check query:

```
SELECT b.book_id,
       b.number_borrowed,
       COALESCE(SUM(br.number), 0) AS actual_number_borrowed
FROM   Book AS b
LEFT JOIN Borrowed AS br ON br.book_id = b.book_id
GROUP BY b.book_id, b.number_borrowed
HAVING b.number_borrowed <> COALESCE(SUM(br.number), 0);

-- Returns (book_id, number_borrowed, actual) only for inconsistent books.
```

(b) ER redesign with Copy_Of entity:

ER Design
Entities: Book, Copy_Of (one instance per physical copy), Person

Relationships:
  Is_Copy_Of: Copy_Of -->1 Book (many-to-one: many copies per book)
  Borrowed_By: Copy_Of -->1 Person (many-to-one: a copy is borrowed by one person)
               [optional: a copy may not be borrowed]

Each Copy_Of has a unique copy_id (synthetic key).

Cardinality ensures consistency: number_borrowed = COUNT of Copy_Of instances related via Borrowed_By. No separate number_borrowed field needed.
number_owned = COUNT of all Copy_Of instances Is_Copy_Of a given Book.

(c) Two relational implementation options:

```
Option 1 (clean, separate table for loan):
  Book(book_id, title)
  Copy(copy_id, book_id)   -- book_id FK to Book
  Loan(copy_id, person_id) -- copy_id FK to Copy, person_id FK to Person
  Person(person_id, name, address)

Option 2 (embed borrower in copy table, NULL if not borrowed):
  Book(book_id, title)
  Copy(copy_id, book_id, person_id)  -- person_id nullable FK to Person
  Person(person_id, name, address)
```

(d) Reproduce original Book table from Option 1:

```
SELECT b.book_id,
       b.title,
       COUNT(c.copy_id)                  AS number_owned,
       COUNT(l.copy_id)                  AS number_borrowed
FROM   Book AS b
LEFT JOIN Copy AS c ON c.book_id = b.book_id
LEFT JOIN Loan AS l ON l.copy_id = c.copy_id
GROUP BY b.book_id, b.title;
```

2022 Paper 3 Q1 — Romantic Comedies & Co-actors

Schema
movies(movie_id, ...), people(person_id, ...), genres(genre_id, ...)
has_genre(movie_id, genre_id), plays_role(movie_id, person_id, role)

(a) Count of romantic comedies:

```
SELECT COUNT(DISTINCT hg1.movie_id) AS total
FROM   has_genre AS hg1
JOIN   genres    AS g1 ON g1.genre_id = hg1.genre_id AND g1.genre = 'Romance'
JOIN   has_genre AS hg2 ON hg2.movie_id = hg1.movie_id
JOIN   genres    AS g2 ON g2.genre_id = hg2.genre_id AND g2.genre = 'Comedy';
```

(b) Co-actor pairs in romantic comedies:

```
SELECT R1.person_id AS pid1,
       R2.person_id AS pid2,
       M.movie_id   AS movie_id
FROM   plays_role AS R1
JOIN   plays_role AS R2 ON R2.movie_id = R1.movie_id
                       AND R2.person_id <> R1.person_id
JOIN   movies     AS M  ON M.movie_id = R1.movie_id
```

```
JOIN   has_genre  AS hg1 ON hg1.movie_id = M.movie_id
JOIN   genres     AS g1  ON g1.genre_id = hg1.genre_id AND g1.genre = 'Romance'
JOIN   has_genre  AS hg2 ON hg2.movie_id = M.movie_id
JOIN   genres     AS g2  ON g2.genre_id = hg2.genre_id AND g2.genre = 'Comedy';
```

(c) name1–title1–name2–title2–name3 chain:

```
SELECT P1.name AS name1, M1.title AS title1,
       P2.name AS name2, M2.title AS title2,
       P3.name AS name3
FROM   plays_role AS PR1  -- name1 in movie1
JOIN   plays_role AS PR2  ON PR2.movie_id = PR1.movie_id -- name2 in movie1
                     AND PR2.person_id <> PR1.person_id
JOIN   plays_role AS PR3  ON PR3.person_id = PR2.person_id -- name2 also in movie2
                     AND PR3.movie_id <> PR1.movie_id
JOIN   plays_role AS PR4  ON PR4.movie_id = PR3.movie_id -- name3 in movie2
                     AND PR4.person_id <> PR2.person_id
JOIN   people AS P1 ON P1.person_id = PR1.person_id
JOIN   people AS P2 ON P2.person_id = PR2.person_id
JOIN   people AS P3 ON P3.person_id = PR4.person_id
JOIN   movies AS M1 ON M1.movie_id  = PR1.movie_id
JOIN   movies AS M2 ON M2.movie_id  = PR3.movie_id
-- Plus the has_genre/genres joins for each movie being a romantic comedy (as above)
-- Plus: name1 NOT in movie2, name3 NOT in movie1
WHERE  NOT EXISTS (SELECT 1 FROM plays_role
                   WHERE person_id = PR1.person_id AND movie_id = PR3.movie_id)
AND    NOT EXISTS (SELECT 1 FROM plays_role
                   WHERE person_id = PR4.person_id AND movie_id = PR1.movie_id);
```

2023 Paper 3 Q1 — Vehicles, GROUP BY, Eventual Consistency

(a)(i) Vehicle ER diagram — key entities and relationships:

ER Design
Entities: Vehicle, Engine, Supplier, PaintColour

Relationships:
Vehicle -->Has_Engine--> Engine (many-to-one: many vehicles can have same engine)
Engine -->Supplied_By--> Supplier (many-to-one, dependent on fuel+hp)
Vehicle <>Available_In<> PaintColour (many-to-many: some models in multiple colours)

Attributes:
Vehicle: vehicle_id (key), model, num_seats
Engine: engine_id (key), fuel_type (petrol/electric), horsepower

Supplier: supplier_id (key), name
PaintColour: colour_id (key), colour_name

Key constraint: supplier is determined by (fuel_type, horsepower) — so Engine has a compound attribute or FK to Supplier keyed on (fuel, hp).

(b)(i) Why GROUP BY data must pass through a reduction operator:

After GROUP BY, each group contains multiple rows — not one. You can only return one value per group in the final result. A reduction operator (aggregate function) collapses those multiple values down to one. Without it, the DBMS would not know which row's value to return for non-grouped columns.

(b)(ii) Required mathematical properties of a reduction operator:

1. Produces a single output value from a multiset of input values.
2. Should be order-independent (commutative, associative) so that rows can be processed in any order or in parallel — enabling query optimisation.
3. Must be well-defined on multisets (not just sets) — e.g. AVG must handle duplicate values correctly, not deduplicate them.

Examples: SUM, COUNT, MIN, MAX, AVG all satisfy these properties.

(c) Eventual consistency — definition and advantages:

Definition: If all update activity ceases, the database will eventually reach a consistent state where all nodes/replicas see the same data.
During updates, different replicas may temporarily hold different values.

Advantages:
1. Higher availability — every replica can respond immediately without waiting for global consensus, even during network partitions.
2. Higher write throughput — no blocking to synchronise all replicas before confirming a write completes.
3. Lower latency for geographically distributed systems — local reads/writes without round-trip to a master node.

(d) Principal disadvantage of eventual consistency + example:

Disadvantage: Reads may return stale or inconsistent data — different clients may see different values for the same data at the same time.

Example: An online shopping cart. User A adds item X and removes item Y in store A. User B (accessing a different replica) still sees item Y in the cart and item X absent. If B checks out before consistency propagates, the order is wrong.

More extreme: two ATMs both read balance £100, both allow £80 withdrawal, resulting in an overdrawn account — ACID is required here.

(e) Why a graph database holds just one graph:

A graph database's power comes from traversing across all node and edge types in a single query. Splitting data into multiple graphs would prevent cross-type path queries (you couldn't follow a chain Movie→Person→Movie without both being in the same graph). The single graph is essentially an in-memory instantiation of the entire ER diagram, with all entities and all relationships coexisting so any path query across any combination of types is efficiently supported.

2024 Paper 3 Q1 — RA Theory, Sizes, ER Design

(a) Does relational union require schemas to share attribute names?

YES — relational union requires both relations to have the SAME set of attribute names (and compatible domains). This is necessary because the result must have a well-defined schema. If the attribute names differ, it is undefined which column should be merged with which. The same applies to intersection — both require matching schemas.

(Note: Some SQL dialects allow UNION with positional matching rather than named matching, but in the formal relational algebra, name matching is required.)

(b) Min/max sizes for union and natural join:

```
Union (P and Q records):
  Minimum: max(P, Q)  — if one relation is a subset of the other
  Maximum: P + Q      — if the two relations are completely disjoint

Natural Join (P and Q records):
  Minimum: 0          — if no tuples share matching values on the join attribute(s)
  Maximum: P × Q      — if all tuples have the same join-attribute value
  (For a foreign key join: exactly P records, assuming all FKs are satisfied)
```

(c) Textbook chapters ER model + relational schema (2 tables, 7 attributes):

Answer

```
ER Diagram:
  Entity: Book   (book_id, title, author, publisher, year)
  Entity: Chapter (chapter_num, chapter_title, page_count)
  Relationship: Contains (Book --> Chapter, one-to-many)

Relational Schema:
  Book(book_id, title)         -- book_id is primary key (synthetic)
  Chapter(book_id, chapter_num, chapter_title) -- (book_id, chapter_num) is composite
PK
                               -- book_id is FK to Book

Total attributes: book_id, title, chapter_num, chapter_title = 4 unique names
(with book_id appearing in both tables as FK — that's 5 columns total, 4 unique names)

For exactly 7 distinct attribute names, expand: e.g.
  Book(book_id, title, author, publisher, year)  -- 5 attributes, book_id is PK
```

Chapter(book_id, chapter_num, chapter_title) -- 3 attributes, (book_id,chapter_num) PK

Keys present: Primary key in Book (simple/natural or synthetic); composite key in Chapter.
Foreign key: Chapter.book_id references Book.book_id (referential integrity).

2024 Paper 3 Q2 — CAP, IsA Hierarchies, Schemas

(a) CAP theorem — complete and define:

C = Consistency: All reads return data that is up-to-date
A = Availability: All clients can find some replica of the data (always responds)
P = Partition Tolerance: System operates despite arbitrary message loss or node failure

Why impossible to achieve all three:
During a network partition, two nodes are disconnected. If we demand Availability, both must respond to reads/writes. If we demand Consistency, they must agree on the current value. But with a partition they cannot communicate to agree.
Contradiction: you must sacrifice either C or A during a partition.
Since partitions can always occur in a real distributed system, P is non-negotiable.
Hence: choose CP (consistent but may be unavailable) or AP (available but may be stale).

(b) Reflexive relation in discrete maths vs. database relation:

Discrete maths: A reflexive relation R on set S means $\forall x \in S, (x,x) \in R$.
Example: ≤ on integers is reflexive because $n \leq n$ for all n.

Database (endorelation): A table relating a domain to itself, e.g.
Manages(manager_id, employee_id) — both columns from the same People domain.
This is NOT automatically reflexive — (x,x) need not appear.

Key difference: The mathematical definition of reflexivity requires (x,x) for ALL x.
A database table only stores the pairs explicitly present — there is no obligation to include self-pairs.

One-to-one cardinality:
Math: a 1-to-1 reflexive relation on S would require every element to relate to exactly itself — only the identity relation works, which is trivial.
Database: a 1-to-1 endorelation can usefully encode a pairing/matching, e.g. a 'buddy_of' relation where each person has exactly one buddy. Rarely useful though.

(c) Schema rearrangement for E→F functional dependency:

Given: R3(A, B, D, E, F) where F is always predictable from E, but costly to compute.

Updates are much rarer than reads.

Since F depends on E (E→F), F is redundant — it violates the normalisation principle.
But: updates are rare, so we tolerate some redundancy for read speed.

Rearrangement: Extract E→F into its own lookup table:
```
EF(E, F)          -- precomputed, keyed on E
R3(A, B, D, E)    -- E is FK to EF
```

Benefit: Reads that need F get it cheaply with a single join on E.
Cost: On update, both R3 and EF must be updated consistently.
Since updates are rare, this cost is acceptable.

(d) Three ways to implement a two-level IsA hierarchy:

Suppose entity Mammal, with sub-entities Lion and Whale.

Way 1 — Three tables:
```
Mammal(mammal_id, name, ...common attributes...)
Lion(mammal_id, mane_colour, ...)   -- FK to Mammal
Whale(mammal_id, whale_type, ...)   -- FK to Mammal
```
Clean. Avoids NULLs. Requires joins to get all data.

Way 2 — Two tables (flatten one sub-entity into parent):
```
MammalAndLion(mammal_id, name, type, mane_colour, whale_type, ...)
```
where whale_type is NULL for lions, mane_colour NULL for whales.
Plus Lion/Whale indicator column. Simpler but introduces NULLs.

Way 3 — One table (fully flattened):
```
AllMammals(mammal_id, type, name, mane_colour, whale_type, ...)
```
All specific attributes present with NULLs for irrelevant type.
Simple to query but most NULLs, most redundancy risk.

For MULTI-LEVEL hierarchies, Way 1 (three-table/per-level) scales best:
Each level adds one table. Queries use joins up the hierarchy chain.
Flattening into one table becomes impractical with many levels.

2025 Paper 3 Q1 — Set Operations, Lists/Sets, Outer Joins

(a) Is the SQL correct for X ∩ (Y ∪ Z)?

Answer
Given SQL:
```
SELECT X.A FROM X JOIN Y JOIN Z WHERE X.A=Y.A OR X.A=Z.A
```

This is INCORRECT. Problems:

1. 'JOIN Y JOIN Z' without ON produces a cross join X × Y × Z, not a union.
2. The WHERE clause then filters, but this can produce DUPLICATE rows
   (if X.A matches both Y.A and Z.A for different Y/Z rows).
3. SQL JOIN without ON is a CROSS JOIN, not a union.

Correct SQL:

```
SELECT A FROM X
WHERE A IN (SELECT A FROM Y)
   OR A IN (SELECT A FROM Z);
```

Or equivalently:

```
SELECT X.A FROM X
WHERE EXISTS (SELECT 1 FROM Y WHERE Y.A = X.A)
   OR EXISTS (SELECT 1 FROM Z WHERE Z.A = X.A);
```

Or using set operations (if X, Y, Z have same schema):

```
SELECT A FROM X
INTERSECT
(SELECT A FROM Y UNION SELECT A FROM Z);
```

(b) Lists vs Sets in CS:

Two differences between a set and a list:
1. Ordering: A list has a defined order (position matters: [A,B] ≠ [B,A]);
   a set has no order ([A,B] = [B,A]).
2. Duplicates: A list can contain duplicates ([A,A,B] is valid);
   a set cannot contain duplicates ({A,B} = {A,A,B}).

(b)(ii) rDBMS schemas for a list and a set:

List schema (ordered, allows duplicates):

```
List(position INTEGER, value TEXT)  -- position is key (or part of key)
PRIMARY KEY (position)
```

The integer position encodes order. Duplicates allowed in 'value' column.

Set schema (unordered, no duplicates):

```
Set(value TEXT)  -- value IS the key (uniqueness enforced)
PRIMARY KEY (value)
```

No position column. Uniqueness constraint prevents duplicates.

(b)(iii) Foo(N,T,A) vs Bar(N,T,A) for list-of-sets, set-of-lists, list-of-lists:

Interpretation: A = item, N = natural number (position/index), T = set/list name.

Foo(N, T, A) — N is key-part, T is key-part, A is value:
If PRIMARY KEY (T, N): T names a list, N is position within T, A is item.
This represents a SET of named lists (each list T can be independently ordered by N).
Cannot represent a list-of-sets because there's no ordering on the set-names T.

Bar(N, T, A) — N is key-part (outer ordering), T is the set name:
If PRIMARY KEY (N, T): N gives the outer list position, T gives the set-name at that position. A is an item in set T. But which items are in set T is a different concern.
Tricky — Bar as stated mixes list-position (N) with set-membership (T, A).

Can either represent a list of lists?
Foo with an outer position (N) indexing into named lists (T) where each T is ordered by a sub-position: need a 4-tuple (outer_pos, list_name, inner_pos, item).
Neither Foo nor Bar as given (3 columns) can cleanly represent a list-of-lists without conflating the two dimensions of ordering.

(c)(i) Example where left outer join is useful:

Scenario: List all students and their exam mark (if they took the exam).

```
SELECT S.name, E.mark
FROM   Students AS S
LEFT OUTER JOIN Exam AS E ON E.student_id = S.id;
```

Students who did not take the exam appear with mark = NULL.
An INNER JOIN would exclude non-participating students entirely.

Non-commutativity: swapping S and E (right outer join) would instead list all exam records, including any orphaned marks with NULL student names.

(c)(ii) Equijoin definition (two-valued logic):

$R \bowtie_{1=1} S$ (equijoin on attribute A from R matching attribute B from S)

$= \{ t \mid \exists u \in R, \exists v \in S,\ u.A = v.B \ \wedge \ t = u \cup v \}$

Or in RA: $\pi_{all}(\sigma_{\{R.A = S.B\}}(R \times S))$

For natural join (A and B are the same attribute name):
$R \bowtie S = \{ t \mid \exists u \in R, \exists v \in S,\ u.[B] = v.[B] \ \wedge \ t = u.[A] \cup u.[B] \cup v.[C] \}$

(c)(iii) Left outer join for three-valued logic:

Given R(A, B, C) and S(B, D) joining on B:

$$R \bowtie S = (R \bowtie S) \cup \{ u \cup (\omega, \omega) \mid u \in R \wedge \neg\exists v \in S, u.B = v.B \}$$

where (ω, ω) is a tuple of NULLs with length equal to the number of attributes in S that are NOT in R — here just D, so one NULL.

Meaning: Include all rows from R joined with matching S rows, PLUS
all R rows with no match in S, padded with NULL for S's non-shared attributes.

2025 Paper 3 Q2 — Vineyard & Recipes

(a)(ii) Schema hardcoding limit of 3 grape types:

```
Wine(wine_id, name, grape1_id, grape1_ratio, grape2_id, grape2_ratio,
    grape3_id, grape3_ratio, bottle_id)
Grape(grape_id, variety_name, ...)
Bottle(bottle_id, shape, volume, ...)

Consistency rules:
  1. Ratios must sum to 1.0: grape1_ratio + COALESCE(grape2_ratio,0)
           + COALESCE(grape3_ratio,0) = 1.0
  2. If grape2_id IS NULL then grape2_ratio IS NULL (and grape3 too)
  3. grape1_id NOT NULL (at least one grape required)
  4. All grape_id values must be FKs into Grape table
```

(a)(iii) Schema without limit on grape types:

```
Wine(wine_id, name, bottle_id)   -- FK bottle_id → Bottle
WineGrape(wine_id, grape_id, ratio)  -- all-key, both FKs
  PRIMARY KEY (wine_id, grape_id)
  FK wine_id → Wine, FK grape_id → Grape
Grape(grape_id, variety_name)
Bottle(bottle_id, shape, volume)

Constraint: SUM(ratio) per wine_id = 1.0 (enforced via trigger or app logic)
```

(a)(iv) Graph database schema for the unrestricted case:

```
Nodes:  (:Wine {wine_id, name})
        (:Grape {grape_id, variety})
        (:Bottle {bottle_id, shape})

Edges:  (:Wine)-[:USES_GRAPE {ratio: 0.6}]->(:Grape)
        (:Wine)-[:BOTTLED_IN]->(:Bottle)

Example:
  CREATE (w:Wine {name: 'Rosé'})
  CREATE (g1:Grape {variety: 'Grenache'})
  CREATE (g2:Grape {variety: 'Syrah'})
  CREATE (w)-[:USES_GRAPE {ratio: 0.7}]->(g1)
  CREATE (w)-[:USES_GRAPE {ratio: 0.3}]->(g2)
```

(a)(v) Relative merits of three schemas:

```
Hardcoded (3 grapes max):
  + Simple to query — no join needed to find all grapes of a wine
```

+ Easy to enforce ratio constraint via CHECK
− Inflexible — breaks if any wine ever needs 4+ grapes
− NULLs for unused grape slots

Relational (no limit):
+ Flexible, normalised, correct referential integrity
+ Handles any number of grapes per wine
− Need a join to retrieve all grapes for a wine
− Ratio sum = 1.0 hard to enforce purely in SQL (needs trigger)

Graph database:
+ Natural representation — grape proportions as edge properties
+ Path queries across wines sharing grapes are simple
− Less natural for sum-of-ratios integrity checks
− Overkill if the main queries are simple lookups

(b)(i) Recipe card storage: (name, text) vs with JSON ingredients:

Option A: (recipe_name, recipe_text) only:
+ Simple. Full flexibility of natural language.
− Ingredients buried in prose — cannot SQL-query 'all recipes using eggs'
− Cannot filter by allergy, cooking time, etc.

Option B: (recipe_name, recipe_text, ingredients_json):
+ Ingredients structured and queryable
+ Original text preserved for context/method
− JSON schema inconsistency across recipes (typed differently by different cooks)
− Adding JSON doesn't help if ingredients are buried in recipe_text too
− Need JSON expansion operators to query

(b)(ii) JSON expansion operations and remaining problems:

Two useful JSON expansions:

1. json_each / JSON_TABLE (lateral flatten / explode):
Expands a JSON array field into multiple rows, one per array element.
E.g.: ingredients_json = ['egg', 'flour', 'butter']
json_each() produces three rows: (recipe, 'egg'), (recipe, 'flour'), (recipe, 'butter')
Useful for: SELECT * FROM recipes, json_each(ingredients_json) WHERE value='egg'

2. JSON path extraction (->> or JSON_EXTRACT):
Extracts a scalar value at a specific path in the JSON.
E.g.: ingredients_json->>'cooking_time' returns '30 mins'
Useful for: SELECT name FROM recipes WHERE ingredients_json->>'cooking_time' < '60 mins'

Problems that remain:

- If ingredients_json has inconsistent structure (e.g. sometimes array, sometimes object)
  queries break. Need COALESCE/CASE to handle variations.
- Type coercion: all JSON values are strings — comparisons like < need explicit CAST.
- Multiple JSON fields each need their own expansion — can't join expansions cleanly.
- Performance: JSON parsing at query time is slow vs. properly indexed columns.
- No foreign key integrity on JSON values.

11 Exam Technique & Common Patterns

11.1 SQL Writing Checklist

Before writing any SQL query:

1. Identify the tables involved and their keys.
2. Identify the join conditions (foreign key relationships).
3. Decide if you need SELECT or SELECT DISTINCT.
   (SELECT produces a multiset; DISTINCT deduplicates.)
4. If counting paths/combinations, think about what each row of the join represents.
5. For anti-joins ('not related to'), use NOT IN subquery or LEFT JOIN IS NULL.
6. For aggregation, decide GROUP BY columns first, then the aggregate.
7. Check: is there a NULL issue? Could any join column be NULL?
8. Alias your tables clearly when doing self-joins.

11.2 ER Diagram Drawing Checklist

1. Read the problem carefully — identify all nouns (entities) and verbs (relationships).
2. Identify the cardinalities — ask 'can one X relate to many Ys?' for each relationship.
3. Decide if each property should be an attribute or a sub-entity.
   Rule: if it has its own properties or can appear multiple times → make it an entity.
4. Are there any weak entities? (existence depends on another entity)
5. Are there IsA / specialisation relationships?
6. Choose appropriate primary keys — prefer synthetic keys.
7. Label all cardinalities (arrows for one-to-many).
8. Check: does each relationship have attributes? If so, add ovals to the diamond.

11.3 Common Exam Question Types and Approaches

Question Type	Key Approach
Draw an ER diagram	Use rectangles, ovals, diamonds, arrows, underline keys. State cardinalities. Consider weak entities and IsA.
Write SQL query	State tables, joins, conditions. Be precise about DISTINCT vs. not. Alias self-joins.
Analyse query output	Think about what each row means. Count paths. Distinguish SELECT vs DISTINCT effects.
Prove/disprove a query's conclusion	The 2021 Q2 style: is the conclusion valid? Look for overcounting (no GROUP BY, self-join duplicates).

Discuss schema design trade-offs	Always mention normalisation vs. redundancy, read vs. write trade-off, ACID vs. BASE.
CAP / consistency model	Define all three letters precisely. Give example of why they conflict.
Eventual consistency	Give advantages (availability, throughput) AND disadvantages (stale reads, no strong guarantee).
Document vs. relational	Discuss joins, updates, consistency, schema flexibility, distribution.

11.4 Classic SQL Mistakes to Avoid

Common SQL Mistakes

1. CROSS JOIN without ON — accidentally gets the Cartesian product.
   Fix: always specify ON condition in JOIN.

2. Forgetting DISTINCT produces duplicates — in path-counting problems use SELECT (not DISTINCT).

3. NULL comparisons — WHERE col = NULL is always NULL (never TRUE).
   Use WHERE col IS NULL instead.

4. NOT IN with NULLs — if the subquery returns any NULL, NOT IN returns empty!
   Safer: NOT EXISTS or LEFT JOIN IS NULL.

5. GROUP BY without including all non-aggregate select columns.
   All selected columns must be either in GROUP BY or inside an aggregate.

6. Self-join confusion — when joining a table to itself, always use aliases (AS a, AS b) and be precise about which alias plays which role.

7. Counting paths vs. distinct pairs — COUNT(*) counts paths (including duplicates).
   For distinct pairs: COUNT(DISTINCT col1, col2) or use subquery with DISTINCT.

11.5 Key Facts to Memorise

Concept	Content	Exam Relevance
ACID properties	Atomicity, Consistency, Isolation, Durability	All four — know each definition
BASE properties	Basically Available, Soft state, Eventual consistency	Contrast with ACID
CAP theorem	Consistency, Availability,	Can only guarantee 2 of 3

	Partition tolerance	
RA operators	σ, π, ρ, ∪, −, ×, ∩, ⋈	Know each one and its SQL equivalent
Transitive closure	$R^+ = \cup\{R^n, n \geq 1\}$	Cannot compute in plain RA; needs recursion
NULL rule	NULL = NULL is NULL, not TRUE	Use IS NULL, not = NULL
Index trade-off	Faster reads, slower writes	log n lookup vs. linear scan
Normalisation rule	All data depends on the key	Split off data depending on non-key fields
Graph DB reason	One graph for cross-type traversal	Splitting graphs breaks path queries
Document update cost	Must update all denormal copies	Name in 1000 documents = 1000 updates

12 Quick Reference Glossary

Term	Definition
Aggregate-oriented DB	Document database — stores data as denormal documents rather than normalised tables
All-key table	A table where the entire row is the key — common for relationship tables
Bacon/Erdős number	Shortest path distance in a co-actor (or co-authorship) graph
Composite key	A key formed by combining two or more attributes
Crow's foot notation	Diagrammatic notation for relationship cardinalities
Denormalisation	Intentional introduction of redundancy for read performance
Discriminator	Partial key attribute for a weak entity
Domain	The set of allowable values for an attribute
Endorelation	A relation from a domain to itself (e.g. co-actor, sibling)
ETL	Extract, Transform, Load — pipeline from OLTP to OLAP warehouse
Functional dependency	A → B: value of A determines value of B
Idempotent	Repeating the operation has no additional effect
Implicit join	SELECT * FROM R, S — equivalent to CROSS JOIN
Inverted index	Index from values to the records containing those values
OLAP	Online Analytical Processing — read-heavy, historical, denormal
OLTP	Online Transaction Processing — update-heavy, current, normalised
Reflexive relation	Mathematical: (x,x) ∈ R for all x. Database: self-pairs not required
Sharding	Horizontally partitioning data across multiple machines by key hash
Superkey	Any set of attributes that uniquely identifies a

	row (not necessarily minimal)
Synthetic key	An artificial key (e.g. auto-increment ID) with no real-world meaning
Transitive closure R^+	Smallest relation containing R that is also transitive
Ternary relationship	A relationship connecting three entities simultaneously
Value atomicity / 1NF	Each field holds exactly one atomic value — no lists or sets in a single field
View	A named stored query — appears as a virtual table to other queries
Weak entity	Entity whose existence depends on another entity; needs discriminator + owner key to be identified

Good luck in the exam!

Cambridge Part IA Databases • Paper 3