Database Management Systems

Optimization, distributed databases and NoSQL

The notes for the lesson “Optimization, distributed databases and NoSQL”: chapters 6.7 to 6.9, with their figures, worked examples, exam traps, quizzes and exam question bank, in one file.

3 chapters | Chapters 6.7 to 6.9

About these notes

These are the notes for the University free n easy video lesson **Optimization, distributed databases and NoSQL** on Database Management Systems: chapters 6.7 to 6.9, with every figure, worked example, exam trap and quiz, and the long-answer exam questions on these chapters.

6.7 Query optimization: cost estimation and heuristics

Query optimization chooses, among the many equivalent ways to evaluate a query, one with a low estimated cost. It works in two ways at once: heuristic rewriting of the query tree, such as doing selections before joins, and cost-based choice of join order and algorithms using statistics kept in the catalog. A good optimizer turns hours into seconds without the user changing a word of the SQL.

You need: Chapter 6.6, evaluation plans and the cost of each algorithm; an earlier lesson, Set operations, Cartesian product and joins in algebra.

Equivalence rules

Two relational algebra expressions are **equivalent** if they give the same rows on every legal database. The optimizer rewrites a query with rules such as these, where E, E_1, E_2 are expressions, θ a condition and L a list of attributes:

1. Cascade of selections: $\sigma_{\theta 1 \wedge \theta 2}(E) = \sigma_{\theta 1}(\sigma_{\theta 2}(E))$.
2. Selections commute: $\sigma_{\theta 1}(\sigma_{\theta 2}(E)) = \sigma_{\theta 2}(\sigma_{\theta 1}(E))$.
3. Cascade of projections: $\pi_{L1}(\pi_{L2}(E)) = \pi_{L1}(E)$ when $L1 \subseteq L2$.

4. A selection over a product is a join: $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$.
5. A selection moves below a join: $\sigma_\theta(E_1 \bowtie E_2) = \sigma_\theta(E_1) \bowtie E_2$ when θ uses only attributes of E_1.
6. Joins commute and associate: $E_1 \bowtie E_2 = E_2 \bowtie E_1$, and $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$.
7. A projection moves below a join if it keeps the join attributes.

Heuristic optimization

1. Split each conjunctive selection into a cascade of single selections (rule 1).
2. Move each selection as far down the tree as it goes (rules 2 and 5), so rows are thrown away early.
3. Combine a Cartesian product with the selection above it into a join (rule 4).
4. Order the leaves so the most restrictive selections and joins, those with the smallest results, run first (rule 6), and avoid any Cartesian product.
5. Push projections down, keeping the attributes later operators need (rules 3 and 7).
6. Mark the subtrees that can be pipelined.

Heuristics are cheap and nearly always help, but they cannot tell which of two join orders is better. For that the optimizer needs numbers.

Cost-based optimization

The catalog keeps statistics for every relation r: n_r (rows), b_r (blocks), l_r (bytes per row), f_r (rows per block), V(A, r) (number of distinct values of attribute A), the minimum and maximum of A, and often a histogram of A. From these the optimizer estimates the size of each intermediate result, assuming values are spread uniformly and conditions are independent:

EXPRESSION	ESTIMATED ROWS
$\sigma_{A = v}(r)$	n_r / V(A, r)
$\sigma_{A \le v}(r)$	$n_r \times (v - \min A) / (\max A - \min A)$
$\sigma_{\theta 1 \wedge \theta 2}(r)$	$n_r \times (s_1 / n_r) \times (s_2 / n_r)$, where s_i rows meet θ_i
r ⋈ s, A a key of r and a foreign key in s	n_s (each row of s meets one row of r)
r ⋈ s on A, general	$n_r \times n_s$ / max(V(A, r), V(A, s))
r × s	$n_r \times n_s$

Each estimate feeds the cost formulas of chapter 6.6, and the optimizer keeps the cheapest plan. The hard part is join order. For n relations there are $(2(n - 1))! / (n - 1)!$ join orders: 12 for 3

relations, 1,680 for 5 and about 17.6 billion for 10. Optimizers since IBM's System R (Selinger and others, 1979) use dynamic programming: find the best plan for every subset of relations once, and build larger plans from those. Many consider only **left-deep trees**, where the right input of every join is a base relation; these suit pipelining and cut the choices for 10 relations to 10! = 3,628,800.



Figure 6.7.1 The worked example's query before and after heuristic rewriting. Orange numbers are estimated rows on each edge. Both trees give the same 500 names, but the rewritten tree never builds the 40-million-row product.

WORKED EXAMPLE

Question. `SELECT c.name FROM customer c, orders o WHERE c.cust_id = o.cust_id AND c.city = 'Chennai' AND o.amount > 5000;` customer has 2,000 rows with V(city) = 20. orders has 20,000 rows; amount is spread uniformly from 0 to 10,000; o.cust_id is a foreign key to customer with V(cust_id, orders) = 2,000. (a) Give the initial query tree and its largest intermediate result. (b) Apply the heuristics. (c) Estimate every intermediate size and the result. `[12 marks]`

1. (a) Direct translation: $\pi_{name}(\sigma_{c.cust_id = o.cust_id \wedge city = 'Chennai' \wedge amount > 5000}(customer \times orders))$. The product has 2,000 × 20,000 = **40,000,000 rows**.
2. (b) Split the selection (rule 1); push city = 'Chennai' onto customer and amount > 5000 onto orders (rule 5); combine the product with c.cust_id = o.cust_id into a join (rule 4): $\pi_{name}(\sigma_{city='Chennai'}(customer) \bowtie \sigma_{amount>5000}(orders))$.
3. (c) $\sigma_{city='Chennai'}(customer)$ = 2,000 / 20 = **100 rows**.
4. $\sigma_{amount>5000}(orders)$ = 20,000 × (10,000 − 5,000) / (10,000 − 0) = **10,000 rows**.
5. Join, assuming all 2,000 customers still appear among the large orders: 100 × 10,000 / max(100, 2,000) = 1,000,000 / 2,000 = **500 rows**. Sense check: 10,000 large orders over 2,000 customers is 5 each, and 100 Chennai customers × 5 = 500.
6. The projection keeps 500 names. The largest intermediate result falls from 40,000,000 rows to 10,000, a 4,000-fold cut, before any algorithm is even chosen.

EXAM TRAPS

- A condition that uses both relations, such as c.cust_id = o.cust_id, stays at the join; only single-relation conditions are pushed down.
- The size formulas assume uniform, independent values. If most orders come from Chennai the estimate is wrong; histograms fix this.
- The general join estimate divides by the larger V, not the smaller.
- (2(n − 1))! / (n − 1)! counts all join orders; n! counts only left-deep ones.

CHECK YOURSELF

– `REMEMBER` **Name four statistics the catalog keeps for a relation.** `2 marks`

Any four of: n_r (rows), b_r (blocks), l_r (row size), f_r (rows per block), V(A, r) (distinct values of A), min and max of A, histograms.

– REMEMBER **State the rule that lets a selection move below a join.** 2 marks

$\sigma_\theta(E_1 \bowtie E_2) = \sigma_\theta(E_1) \bowtie E_2$ when θ uses only attributes of E_1.

– UNDERSTAND **What is a left-deep join tree, and why do optimizers prefer it?** 4 marks

A join tree in which the right input of every join is a base relation, so the joins form a chain. The growing intermediate result can be pipelined into the next join as the outer input, an index on each base relation can be used, and the number of orders to examine drops from (2(n – 1))!/(n – 1)! to n!.

– APPLY **r has 10,000 rows and V(dept, r) = 25. Estimate the size of $\sigma_{dept='CSE'}(r)$.** 1 mark

10,000 / 25 = **400 rows**.

– APPLY **student has 3,000 rows (roll_no is its key); enrol has 24,000 rows with roll_no a non-null foreign key and V(roll_no, enrol) = 3,000. Estimate student ⋈ enrol both ways.** 3 marks

Key–foreign key rule: each enrol row matches one student, so **24,000 rows**. General formula: 3,000 × 24,000 / max(3,000, 3,000) = **24,000**. They agree.

– APPLY **How many join orders are there for 4 relations, and how many are left-deep?** 2 marks

(2 × 3)! / 3! = 720 / 6 = **120**. Left-deep: 4! = **24**.

– ANALYZE **When can "do selections first" be a poor choice?** 4 marks

When the selection is costly per row (a user-defined function or a pattern match on long text) and a join would first discard most rows, so fewer rows reach the costly test. Also when filtering a relation first throws away its index, so an indexed nested-loop join on the unfiltered relation would have been cheaper. Cost-based optimization catches both cases; pure heuristics do not.

BEYOND THE SYLLABUS

Not for the exam. SQL Server's optimizer follows the Cascades framework, which explores rewrites and costs together. PostgreSQL switches to a genetic search when a query joins very many tables. Spark's adaptive query execution re-plans in the middle of a query once real sizes are known, and research systems learn cardinality estimates from past queries. Search "cardinality estimation" and "Cascades optimizer" next.

6.8 Distributed and parallel databases

When one server is not enough, a database is spread over many machines. A parallel database uses many processors and disks, usually in one data centre, to run queries faster. A distributed database stores data at sites that may be far apart, such as a bank's data centres in Mumbai and Chennai, and lets users query it as one database.

You need: Chapter 6.6, join algorithms; an earlier lesson, ACID properties and Lock-based protocols and two-phase locking.

Parallel databases

There are three architectures (Figure 6.8.1):

- **Shared memory**: all processors share one memory and the disks over a bus. Communication is fastest, but the bus saturates beyond a few tens of processors.
- **Shared disk**: each processor has its own memory, and all reach all disks over an interconnect. If a node fails, another takes over its work because the disks are shared; the disk interconnect becomes the bottleneck. Oracle RAC works this way.
- **Shared nothing**: each node has its own processor, memory and disks and talks to the others only over the network. It scales furthest, because nodes have nothing to contend for; most data warehouses and NoSQL stores use it. A hierarchical design combines them, for example shared-nothing nodes that are each shared-memory machines.

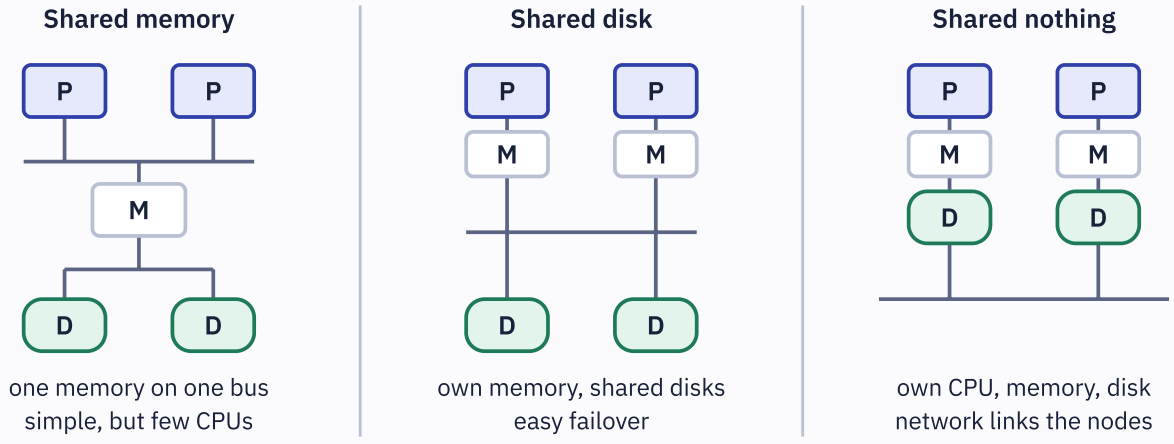

Figure 6.8.1 P = processor, M = memory, D = disk. Moving right, nodes share less: shared memory shares memory and disks, shared disk shares only the disks, and shared nothing shares only the network, which is why it scales furthest.

Data is spread over the disks by **partitioning**: **round-robin** (row i goes to disk i mod n; an even spread, but every lookup must ask every disk), **hash** (a hash of a key picks the disk; an exact match goes to one disk) and **range** (key ranges, such as roll numbers 1 to 1000 on disk 0; good for range queries, but at risk of skew).

Two measures judge a parallel system. **Speedup** = T_{small} / T_{large} for the same task on N times the hardware; ideally N (linear speedup). **Scaleup** = time of a task on the small system ÷ time of an N-times task on the N-times system; ideally 1 (linear scaleup). Start-up costs, interference over shared resources and skew keep real systems below linear. Parallelism comes as **interquery** (many queries at once, for throughput) and **intraquery**: intraoperation, one sort or join split over all nodes, and interoperation, different operators of one plan running at once. A parallel hash join partitions both relations on the join attribute, so each node joins only its own pair of partitions.

Distributed databases

- **Homogeneous** systems run the same DBMS at every site; **heterogeneous** (multidatabase) systems join different DBMSs through a middleware layer.
- **Fragmentation** splits a relation. Horizontal fragments are row subsets chosen by a selection, such as the Chennai branch's accounts stored at Chennai; the relation is rebuilt by UNION. Vertical fragments are column subsets that each keep the key; the relation is rebuilt by a join on the key. Mixed fragmentation does both.
- **Replication** keeps copies at several sites: reads are faster and survive a site failure, but every update must reach every copy.
- **Transparency**: users need not know where fragments are (location), how a relation is split (fragmentation) or how many copies exist (replication).

Distributed query processing counts the bytes shipped over the network, often the largest cost. A **semijoin** $r \ltimes s$ ships only the join column of s to r's site, keeps the rows of r that match, and ships just those back; it pays when few rows match.

A transaction that updates several sites must commit everywhere or nowhere. **Two-phase commit (2PC)** does this. Phase 1: the coordinator sends PREPARE; each participant writes its changes and a ready record to its log and votes READY or ABORT. Phase 2: if every vote is READY, the coordinator logs COMMIT and tells all to commit; otherwise it tells all to abort. Its weakness is **blocking**: a participant that voted READY must wait, holding its locks, if the coordinator fails before sending the decision. Finally, the **CAP theorem** (Brewer's conjecture, proved by Gilbert and Lynch in 2002) says that during a network partition a distributed system must give up either consistency or availability.

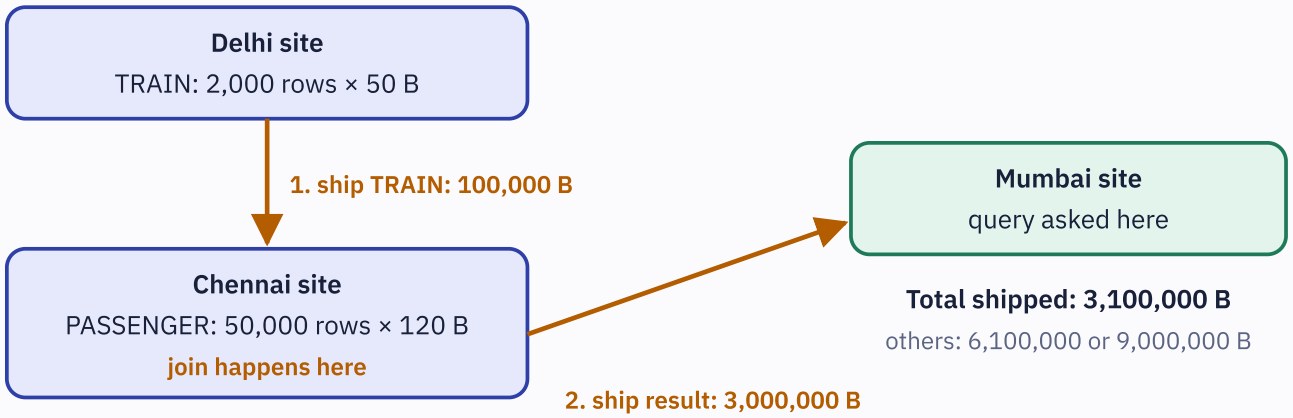

Figure 6.8.2 The cheapest plan of the worked example: move the small relation to the big one, join there, and send only the answer to the user. Moving the 6,000,000-byte PASSENGER table anywhere is what the other two plans pay for.

WORKED EXAMPLE

Question. A railway database stores PASSENGER(pnr, name, train_no, ...) at Chennai: 50,000 rows of 120 bytes. TRAIN(train_no, train_name, ...) is at Delhi: 2,000 rows of 50 bytes. A user at Mumbai asks for every passenger's name with the train name. Each result row is 60 bytes and every passenger matches one train. (a) Compare three strategies by bytes shipped. (b) Which is best if the query is asked at Chennai? (c) A sort takes 100 minutes on one node and 12 minutes on 10 nodes: find the speedup. `[10 marks]`

1. Sizes: PASSENGER = 50,000 × 120 = 6,000,000 B; TRAIN = 2,000 × 50 = 100,000 B; result = 50,000 × 60 = 3,000,000 B.
2. Strategy 1, ship both relations to Mumbai and join there: 6,000,000 + 100,000 = **6,100,000 B**.
3. Strategy 2, ship PASSENGER to Delhi, join, ship the result to Mumbai: 6,000,000 + 3,000,000 = **9,000,000 B**.
4. Strategy 3, ship TRAIN to Chennai, join, ship the result to Mumbai: 100,000 + 3,000,000 = **3,100,000 B**. Best (Figure 6.8.2).
5. (b) At Chennai, ship TRAIN to Chennai and join; the result is already where it is wanted: **100,000 B**.
6. (c) Speedup = 100 / 12 = **8.33** on 10 nodes, below the linear 10, because of start-up, coordination and skew.

EXAM TRAPS

- Speedup keeps the task fixed and adds hardware; scaleup grows the task and the hardware together.
- Horizontal fragments are rebuilt by UNION, vertical ones by a JOIN on the key, so every vertical fragment must keep the key.
- Two-phase commit (2PC) is not two-phase locking (2PL, an earlier lesson): 2PC makes a distributed transaction atomic; 2PL makes schedules serializable.
- In distributed cost questions, count bytes shipped over the network unless told to count blocks.

CHECK YOURSELF

– REMEMBER **Name the three parallel database architectures.** 2 marks

Shared memory, shared disk and shared nothing (and hierarchical combinations of them).

– REMEMBER **Define horizontal and vertical fragmentation.** 2 marks

Horizontal: a fragment is a subset of rows chosen by a selection; rebuilt by union. Vertical: a fragment is a subset of columns including the key; rebuilt by a join on the key.

– UNDERSTAND **Explain the two phases of 2PC and its blocking problem.** 6 marks

Phase 1 (voting): the coordinator sends PREPARE; each participant forces its log and votes READY or ABORT. Phase 2 (decision): all READY → the coordinator logs and sends COMMIT; any ABORT or timeout → ABORT; participants act and acknowledge. Blocking: a READY participant may neither commit nor abort on its own, so if the coordinator crashes before the decision arrives, it waits with its locks held until the coordinator recovers.

– UNDERSTAND **Which partitioning serves a range query on roll_no best: round-robin, hash or range?** 3 marks

Range partitioning on roll_no: only the disks whose ranges overlap the query are read. Round-robin and hash scatter neighbouring roll numbers, so every disk must be searched.

– APPLY **In the worked example, the same query is asked at Delhi. Find the cheapest strategy.** 3 marks

Ship PASSENGER to Delhi and join there: 6,000,000 B. Ship TRAIN to Chennai, join, and ship the result to Delhi: 100,000 + 3,000,000 = **3,100,000 B**, the cheaper plan.

– APPLY **A report takes 40 minutes on 4 nodes. With 8 nodes and twice the data it takes 44 minutes. Find the scaleup.** 2 marks

Scaleup = 40 / 44 = **0.91**; linear scaleup would be 1.

– ANALYZE **During a network split between its Mumbai and Chennai data centres, should a UPI payment system choose consistency or availability? Would a video "like" counter choose the same?** 4 marks

Payments should choose consistency: refuse or delay payments on the cut-off side rather than let a balance be spent twice. A like counter can choose availability: accept likes on both sides and merge the counts when the link returns, since a briefly wrong count harms nobody.

BEYOND THE SYLLABUS

Not for the exam. Distributed SQL systems such as Google Spanner and CockroachDB replicate each piece of data with consensus protocols (Paxos or Raft), so a commit survives the loss of a site and 2PC's blocking is far rarer. Search "Raft consensus" and "MapReduce join" next.

6.9 NoSQL databases: document, key-value and graph

NoSQL ("not only SQL") is a family of databases that drop the fixed relational schema, and often some ACID guarantees, to scale out across many cheap servers and to store data in the shape the application uses: key-value pairs, JSON documents, wide columns or graphs. They grew from the needs of web companies in the 2000s, such as Amazon's Dynamo and Google's Bigtable, and now run shopping carts, app backends and recommendation engines.

You need: Chapter 6.8, partitioning, replication and CAP; an earlier lesson, ACID properties.

Why NoSQL, and BASE

A relational DBMS gives a fixed schema, joins and ACID transactions, and it grows mainly by moving to a bigger server (scaling up). At web scale that hits limits: one server has a ceiling, joins across machines are slow, and changing the schema of a billion-row table takes hours. NoSQL systems **scale out** instead. They partition (shard) data on a key across many servers, replicate

each shard, keep the schema in the application (schema-on-read), and favour simple queries by key.

Many choose availability over strict consistency in the CAP sense and call themselves **BASE**: Basically Available, Soft state, Eventually consistent. Replicas may disagree for a moment after an update but converge once updates stop. Consistency is often tunable: with N copies of each item, a write waits for W replicas and a read asks R replicas. If **R + W > N**, every read set overlaps every write set, so a read sees the latest write.

TYPE	DATA MODEL	EXAMPLES	GOOD FOR
Key-value	key → value the store does not interpret	Redis, Amazon DynamoDB, Riak	sessions, caches, carts
Document	key → JSON-like document with nested fields	MongoDB, CouchDB	catalogues, content, app backends
Wide-column	row key → column families of many columns	Apache Cassandra, HBase	time series, logs, heavy writes
Graph	nodes and edges, both with properties	Neo4j, Amazon Neptune	social networks, fraud rings, recommendations

Key-value and document stores

A **key-value store** offers get(key), put(key, value) and delete(key). The value is opaque: the store cannot search inside it, so every access needs the key. Redis commands, each followed by the server's reply:

```
SET cart:u42 "dosa:2,coffee:4"     → OK
EXPIRE cart:u42 1800               → (integer) 1
GET cart:u42                       → "dosa:2,coffee:4"
```

The cart now disappears 1,800 seconds (30 minutes) after the EXPIRE command, which suits an abandoned shopping cart.

A **document store** keeps self-describing documents, usually JSON (MongoDB stores a binary form called BSON), grouped in collections. Documents in one collection need not share fields, nested arrays and sub-documents are allowed, and queries and secondary indexes can use any field. The main design choice is **embedding versus referencing**: embed data that is read together and owned by one parent, such as the items of an order; reference data that many documents share or that grows without limit, such as the customer, by cust_id.

Graph databases

A **graph database** stores a property graph: nodes with labels and properties, such as Person {name: 'Asha'}, and directed relationships with a type and properties, such as FRIEND {since: 2021}. Each node keeps direct references to its relationships (index-free adjacency), so following an edge is a pointer hop and a query costs in proportion to the part of the graph it touches. The same question in SQL needs a self-join of a large Friend table for every hop.

Neo4j's query language, Cypher, draws the pattern with arrows. For the graph in Figure 6.9.1:

```
MATCH (a:Person {name: 'Asha'})-[:FRIEND]->()-[:FRIEND]->(f)
WHERE f <> a
RETURN DISTINCT f.name;

f.name
"Deepak"
"Esha"
```

Deepak is reached twice, through Bala and through Chitra; DISTINCT keeps one. Relational databases remain the better choice when the structure is stable, when a transaction must span many records (a bank ledger), and for ad-hoc joins and reports. Many applications use several stores side by side, which is called polyglot persistence.



Figure 6.9.1 Three NoSQL data models. Left: the store sees only keys and opaque values. Middle: order 1001 as one document, with its items embedded. Right: arrows are FRIEND relationships; Asha (orange) reaches Deepak and Esha (green) in two hops, the answer of the Cypher query.

WORKED EXAMPLE

Question. A food-delivery app in Chennai stores orders in MongoDB. (a) Write the document for order 1001: customer 42, Chennai, 2 masala dosas at ₹120 each and 4 filter coffees at ₹100 each, with the total. (b) The collection holds orders 1001 (Chennai, ₹640), 1002 (Chennai, ₹380), 1003 (Madurai, ₹900) and 1004 (Chennai, ₹1,250). Write a query for Chennai orders above ₹500 showing the amount, with its output and SQL equivalent. (c) Each order is kept on N = 3 replicas. Does W = 2, R = 2 guarantee reading the latest write? Does W = 1, R = 1? (d) Pick a store for the cart, the menus, "dishes your friends' friends ordered" and payments. `[12 marks]`

1. (a) Total = 2 × 120 + 4 × 100 = 240 + 400 = **₹640**.

```
{ _id: 1001, cust_id: 42, city: "Chennai", amount: 640,
  items: [ { dish: "Masala dosa",   qty: 2, price: 120 },
           { dish: "Filter coffee", qty: 4, price: 100 } ] }
```

2. (b) The query and its output in the MongoDB shell:

```
db.orders.find({ city: "Chennai", amount: { $gt: 500 } }, { amount: 1 })
[ { _id: 1001, amount: 640 }, { _id: 1004, amount: 1250 } ]
```

   Order 1002 fails the amount test (380 ≤ 500) and 1003 fails the city test. SQL: `SELECT order_id, amount FROM orders WHERE city = 'Chennai' AND amount > 500;` MongoDB returns _id unless told not to.

3. (c) W = 2, R = 2: R + W = 4 > 3, so every read overlaps the latest write: **yes**. W = 1, R = 1: R + W = 2 ≤ 3, so a read may reach a replica not yet updated: **no**.

4. (d) Cart: **key-value** (one lookup by user id, with expiry). Menus: **document** (each restaurant's menu has different fields). Friends-of-friends dishes: **graph** (multi-hop traversal). Payments: **relational** (ACID across accounts).

5. Compare: in a relational design, order 1001 is 1 row in orders plus 2 rows in order_items, read with a join; the document is read in one access.

EXAM TRAPS

- NoSQL means "not only SQL", not "no SQL": Cassandra's CQL and several document stores offer SQL-like languages.
- Eventual consistency is not "no consistency": replicas converge once updates stop.
- Embedding duplicates data. If a restaurant's name is embedded in a million orders, renaming it means updating a million documents.
- R + W > N makes a read see the latest write; it does not give multi-record ACID transactions.

CHECK YOURSELF

– REMEMBER **Name the four NoSQL families with one product each.** 2 marks

Key-value (Redis), document (MongoDB), wide-column (Cassandra) and graph (Neo4j).

– REMEMBER **Expand BASE.** 1 mark

Basically Available, Soft state, Eventually consistent.

– UNDERSTAND **When should a document embed related data, and when should it reference it?** 4 marks

Embed data read together and owned by one parent, with bounded size (order items, an address): one read fetches everything. Reference data shared by many documents or growing without limit (customers, a product's reviews): this avoids duplicate copies to update and documents that grow too large.

– UNDERSTAND **Why is a friends-of-friends query faster in a graph database than with SQL self-joins on a large table?** 4 marks

The graph database starts at one node and follows stored edge references, touching only the friends and their friends. SQL joins the Friend table with itself; even with an index, each hop is an index lookup per row, and the cost grows with the table and the number of hops.

– APPLY **N = 5 replicas and W = 3. Find the smallest R that guarantees reading the latest write.** 2 marks

Need R + 3 > 5, so R > 2: **R = 3**.

– APPLY **On the four sample orders, write a MongoDB query for orders from Madurai or of at least ₹1,000, and give its output.** 3 marks

`db.orders.find({ $or: [ { city: "Madurai" }, { amount: { $gte: 1000 } } ] })` returns **1003** (Madurai, 900) and **1004** (Chennai, 1250).

– ANALYZE **Why should a bank keep account balances in a relational DBMS rather than an eventually consistent key-value store?** 4 marks

A transfer changes two balances, which must change together (atomicity) and never be read half-done. An eventually consistent store may return a stale balance, letting money be spent twice, and has no multi-key transactions. The bank needs ACID, constraints such as balance ≥ 0, and ad-hoc audit queries, which a relational DBMS gives.

BEYOND THE SYLLABUS

Not for the exam. The line between SQL and NoSQL keeps blurring: MongoDB added multi-document ACID transactions in version 4.0 (2018), and PostgreSQL stores and indexes JSON documents. ISO published GQL, a standard query language for property graphs, in 2024. Vector databases, and extensions such as pgvector, store embeddings for the similarity search behind AI assistants. Search "polyglot persistence" and "vector database" next.

Key terms of this lesson

TERM	MEANING OR FORMULA	CHAPTER
Selection size	$\sigma_{A=V}$: n_r / V(A, r)	6.7
Join size	$n_r \times n_s$ / max(V(A, r), V(A, s))	6.7
Speedup and scaleup	speedup = T_{small} / T_{large} (ideal N); scaleup ideal 1	6.8
Quorum rule	R + W > N: every read sees the latest write	6.9

Exam question bank

Long-answer exam questions on these chapters, each with an outline of the points an examiner looks for. Answer one in full on paper before you read its outline.

Remember and understand

- UNDERSTAND **Describe the parallel database architectures, data partitioning and speedup and scaleup. Explain two-phase commit for distributed transactions.** 13 marks

  Points: shared memory, shared disk, shared nothing (and hierarchical) with diagrams and one advantage and limit each; round-robin, hash and range partitioning; speedup and scaleup formulas with ideal values and the reasons for sub-linear results; 2PC's prepare and decision phases, log records, and the blocking problem.

Apply

- APPLY **Student(roll, name, dept): 3,000 rows, V(dept) = 6. Enrol(roll, cid, grade): 24,000 rows, V(grade) = 8, V(roll) = 3,000. Course(cid, title, credits): 300 rows, V(credits) = 4. For "names of CSE students with grade A in a 4-credit course", draw the initial tree, optimize it heuristically and estimate every intermediate size.** 16 marks

  Initial: $\pi_{name}(\sigma_{conditions}(\text{Student} \times \text{Enrol} \times \text{Course}))$, a product of 3,000 × 24,000 × 300 = 21,600,000,000 rows. Heuristic tree: $\pi_{name}((\sigma_{dept='CSE'}\text{Student} \bowtie \sigma_{grade='A'}\text{Enrol}) \bowtie \sigma_{credits=4}\text{Course})$. Sizes: 3,000/6 = 500; 24,000/8 = 3,000; 300/4 = 75. First join on roll: 500 × 3,000 / max(500, 3,000) = 500. Second join on cid: 500 × 75 / max(300, 75) = **125 rows**. Push projections to keep only roll, name, cid. This spans chapters 6.6 and 6.7 and an earlier lesson.

Analyze and evaluate

– EVALUATE **A food-delivery start-up must store orders, carts, restaurant menus, payments and friend-based recommendations, and expects ten times growth in a year. Evaluate one relational server, a distributed relational database and a mix of NoSQL stores, and recommend a design.** 16 marks

Points: what each workload needs (ACID for payments; flexible fields for menus; key lookups with expiry for carts; multi-hop traversal for recommendations); scaling up against scaling out; partitioning orders by city or customer, replication and the CAP trade-off; R + W > N for carts; polyglot persistence and its cost of running several systems; a justified recommendation. Spans chapters 6.8 and 6.9.

– EVALUATE **"Our database runs on RAID 10 and is replicated to a second data centre, so it needs no backups or recovery log." Evaluate this claim.** 8 marks

Points: RAID and replication protect against device and site failure only; a wrong DELETE, an application bug or ransomware is copied to every mirror and replica at once; point-in-time recovery needs backups plus the log (an earlier lesson, Log-based recovery and write-ahead logging); the log also gives atomicity and durability after a crash; asynchronous replication can lose the last committed transactions. Verdict: the claim is false. Spans chapters 6.1 and 6.8 and an earlier lesson.

Glossary (terms used in this lesson)

Secondary index	A dense index on a field that does not set the file order.
Query tree	A relational algebra expression drawn as a tree, with tables at the leaves.
Evaluation plan	A query tree annotated with the algorithm and index for every operation.
Pipelining	Passing rows from one operator to the next as they are produced, without temporary relations.
Left-deep join tree	A join order in which the right input of every join is a base relation.
Shared nothing	A parallel architecture in which each node has its own processor, memory and disks.
Fragmentation	Splitting a relation into row subsets (horizontal) or column subsets (vertical) stored at different sites.
Two-phase commit (2PC)	A protocol that makes a transaction commit at all sites or at none.
CAP theorem	During a network partition, a distributed system must give up consistency or availability.

BASE	Basically available, soft state, eventually consistent: the guarantees many NoSQL stores give instead of ACID.

Sources and further reading

Textbooks

- Abraham Silberschatz, Henry F. Korth and S. Sudarshan, *Database System Concepts*, 7th ed., McGraw-Hill. Chapters on physical storage systems, data storage structures, indexing, query processing, query optimization, and parallel and distributed databases.
- Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems*, 7th ed., Pearson. Chapters on disk storage and file structures, indexing structures, query processing and optimization, distributed databases and NoSQL systems.
- Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*, 3rd ed., McGraw-Hill. Parts on storage and indexing and on query evaluation.
- Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom, *Database Systems: The Complete Book*, 2nd ed., Pearson. Chapters on secondary storage, index structures and query execution.

Original papers

- D. A. Patterson, G. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", ACM SIGMOD, 1988.
- R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica*, 1972.
- R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, "Extendible Hashing: A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems*, 1979.
- W. Litwin, "Linear Hashing: A New Tool for File and Table Addressing", VLDB, 1980.
- P. G. Selinger and others, "Access Path Selection in a Relational Database Management System", ACM SIGMOD, 1979.

Web resources

The writing session's network could not open web pages, so none of these was opened while writing. They are well-known open resources, listed from memory:

- Slides and resources for *Database System Concepts*: db-book.com
- PostgreSQL documentation, chapters "Indexes" and "Using EXPLAIN": postgresql.org/docs/current/indexes.html and postgresql.org/docs/current/using-explain.html

- MongoDB Manual, query documents: mongodb.com/docs/manual
- Neo4j Cypher Manual: neo4j.com/docs/cypher-manual

Syllabus: the University free n easy universal syllabus, built from top universities' syllabi.

Built for the University free n easy YouTube channel from the universal syllabus of Database Management Systems (Oct 2026). Written with top universities' syllabi and standard textbooks; check a formula against your prescribed book before an exam, and report mistakes on the channel.