

VU LEARNING HUB

vulearninghub.xyz

HANDOUT

CS515

Advanced Database Management System

CS515 by VU Learning HUB

Prepared for VU students • Free study resources

<https://vulearninghub.xyz>

COMPREHENSIVE STUDY HANDOUT

CS515

Advanced Database Management System

Lecture-wise study notes prepared for Virtual University of Pakistan students.
Covers all major topics with objectives, key definitions, summaries and exam pointers.

13 study units

Table of Contents

Placeholder for table of contents

0

Unit 1: Beyond Relational: Why Advanced DBMS

Learning Objectives

- Objective 1:** Explain the limitations of classical relational systems

Objective 2: Describe the landscape of advanced data management

Objective 3: Classify advanced DBMS architectures

The classical relational DBMS, dominant since the 1980s, assumes centralized data, structured tables, and ACID transactions on a single machine. Modern applications break every one of these assumptions: web-scale data is distributed across continents, semi-structured formats like JSON and XML dominate, and workloads mix transactions with analytics. Advanced database management studies the systems built for this reality: distributed, parallel, object-oriented, and non-relational engines.

Three forces drove the advance. First, distribution: organizations needed data close to users and resilient to failures, giving distributed DBMS. Second, performance: single machines hit limits, giving parallel DBMS and shared-nothing architectures. Third, data variety: objects, documents, graphs and streams did not fit tables, giving object-oriented DBMS, XML databases and NoSQL systems. An advanced-DBMS professional must match each workload to the right architecture.

This course covers distributed database design (fragmentation, replication, allocation), distributed query processing and optimization, distributed transaction management and concurrency control, distributed deadlock and recovery, object and object-relational systems, semi-structured data management, data warehousing, and parallel database technology. Very important: the unifying theme is preserving correctness (consistency, isolation, durability) while data and processing are spread across machines.

Key Definitions

Distributed DBMS is defined as a DBMS that manages a logically unified database physically spread across multiple networked sites.

ACID is defined as the transaction properties: Atomicity, Consistency, Isolation, Durability.

Shared-nothing architecture is defined as a parallel design where each processor has private memory and disks, communicating only via messages.

Key Points to Remember

- Classical RDBMS assumes centralized, structured, single-machine data; modern apps violate all three.
- Drivers: distribution (resilience/locality), parallelism (performance), variety (objects, documents, graphs).
- Course spine: distributed design, distributed queries, distributed transactions, objects/XML, warehousing, parallelism.
- Unifying theme: correctness guarantees across distributed machines.

Unit Summary

Advanced DBMS extends classical relational systems to handle distributed, parallel and non-relational workloads while preserving correctness guarantees.

Exam Pointer

Very important: be able to explain with examples why a classical centralized RDBMS is insufficient for modern applications. This motivates the entire course and opens many exam papers.

Unit 2: Distributed Database Design: Fragmentation

Learning Objectives

- Objective 1:** Define fragmentation and its objectives
- Objective 2:** Distinguish horizontal, vertical and mixed fragmentation
- Objective 3:** Apply fragmentation correctness rules

Fragmentation partitions a relation into pieces stored at different sites so that each site holds the data it uses most. The objectives are locality of access (queries run where their data lives), reduced communication cost, and improved availability through placement. Fragmentation is a design decision made before allocation (which fragment goes where) and replication (which fragments are copied).

Horizontal fragmentation splits a relation by rows using selection predicates, e.g. employees partitioned by branch city. It comes in primary form (predicate on the relation itself) and derived form (partitioned via a join to another fragmented relation, e.g. orders partitioned by their customer's region). Vertical fragmentation splits by columns, grouping attributes that are accessed together, always keeping the primary key in every fragment so the relation can be reconstructed by join. Mixed fragmentation combines both.

Correctness is governed by three rules. Completeness: every data item appears in some fragment (no loss). Reconstruction: the original relation can be rebuilt (union for horizontal, join for vertical). Disjointness: for horizontal fragmentation, fragments do not overlap (except vertical, where only the key repeats). A design violating these rules loses or duplicates data. Very important: exam problems frequently ask you to fragment a given schema and then verify the three rules.

Key Definitions

Horizontal fragmentation is defined as partitioning a relation into subsets of tuples using selection predicates.

Vertical fragmentation is defined as partitioning a relation into subsets of attributes, keeping the key in each fragment.

Derived horizontal fragmentation is defined as partitioning a relation according to the fragmentation of another relation via a join attribute.

Key Points to Remember

- Goals: locality, less communication, better availability.
- Horizontal = by rows (primary or derived); vertical = by columns (key retained); mixed = both.
- Correctness rules: completeness, reconstruction, disjointness.
- Derived fragmentation follows join paths, e.g. orders by customer region.

Unit Summary

Fragmentation splits relations by rows, columns or both to improve locality; correct designs satisfy completeness, reconstruction and disjointness.

Exam Pointer

Very important: practice fragmenting a sample schema (e.g. EMP by dept, PROJ by city) and verifying the three correctness rules. Derived horizontal fragmentation is a favourite twist.

Unit 3: Replication and Data Allocation

Learning Objectives

Objective 1: Explain replication models and their trade-offs

Objective 2: Compare full, partial and no replication

Objective 3: Describe allocation strategies

Replication stores copies of fragments at multiple sites. Its benefits are compelling: higher availability (data survives site failures), better reliability, and faster local reads. The cost is update propagation: every write must reach all copies, increasing communication and complicating consistency. This read-vs-write trade-off is the central tension of replication design.

Three models span the spectrum. Full replication copies the entire database everywhere: maximum availability and read speed, but every update is global, so it suits read-heavy workloads. No replication stores each fragment once: cheapest updates but a site failure loses its data. Partial replication, the practical middle, copies only hot fragments to the sites that need them. Designers choose per-fragment based on read/write ratios and availability requirements.

Allocation decides which site holds each fragment or copy, optimizing for communication cost, storage limits and processing load. Strategies range from centralized (everything at one site) through partitioned to fully replicated. Modern cloud systems automate much of this, but the underlying optimization, minimizing a cost function over query frequencies, remains the same. Remember: fragmentation decides the pieces, replication decides the copies, allocation decides the addresses.

Key Definitions

Replication is defined as maintaining copies of data fragments at multiple sites.

Read/write ratio is defined as the proportion of read to update operations, which drives replication decisions.

Data allocation is defined as assigning fragments and their copies to specific sites.

Key Points to Remember

- Replication gains: availability, reliability, local reads; cost: update propagation.
- Full replication = max availability, expensive updates; partial = practical middle; none = cheapest, fragile.
- Choose per fragment using read/write ratios and availability needs.
- Fragmentation -> pieces; replication -> copies; allocation -> placement.

Unit Summary

Replication trades read performance and availability against update cost; partial replication is the practical norm, and allocation places fragments to minimize communication.

Exam Pointer

Exam pointer: be ready to argue for a replication level given a workload description (e.g. read-heavy reporting vs write-heavy OLTP). The read/write trade-off is the key insight.

Unit 4: Distributed Query Processing

Learning Objectives

Objective 1: Outline the phases of distributed query processing

Objective 2: Explain data localization and global optimization

Objective 3: Describe join strategies: ship-whole, fetch-as-needed, semi-join

A distributed query passes through four phases. Query decomposition rewrites it into relational algebra on global relations. Data localization replaces global relations with their fragments using fragmentation information. Global optimization chooses the execution plan, deciding join order and data movement across sites. Local optimization then optimizes each site's subquery with local statistics. Cost here is dominated by data transfer, not disk I/O, which flips classical optimization priorities.

The critical decision is where joins execute and what moves. Three classic strategies illustrate the trade-offs. Ship-whole sends entire relations to one site: simple but wasteful. Fetch-as-needed ships only required tuples per outer tuple: good for selective queries but chatty. Semi-join ships only the join attributes first to filter the other relation, then ships just matching tuples: excellent when join selectivity is low. The optimizer estimates sizes using statistics to pick the cheapest.

Distributed optimization is hard because the search space explodes with sites and fragments, and statistics may be stale. Systems therefore use heuristics plus cost models weighting communication heavily. Very important: in exams, when asked to compare strategies, always argue in terms of bytes transferred under stated selectivities; that quantitative reasoning is what earns full marks.

Key Definitions

Data localization is defined as rewriting a query on global relations into an equivalent query on physical fragments.

Semi-join is defined as a join strategy that first ships join attributes to reduce the other relation before shipping full tuples.

Query decomposition is defined as translating a query into relational algebra over global relations.

Key Points to Remember

- Four phases: decomposition, localization, global optimization, local optimization.
- Communication cost dominates; classical I/O-focused optimization does not apply.

- Strategies: ship-whole, fetch-as-needed, semi-join; choice depends on selectivity.
- Argue quantitatively: bytes transferred under given selectivities.

Unit Summary

Distributed query processing rewrites queries onto fragments and optimizes primarily for communication cost, choosing join strategies like semi-join based on selectivity estimates.

Exam Pointer

Very important: practice a numeric comparison of ship-whole vs semi-join given relation sizes and selectivity. Show bytes transferred step by step.

Unit 5: Distributed Transaction Management

Learning Objectives

- Objective 1:** Define distributed transactions and their ACID challenges
- Objective 2:** Explain the two-phase commit protocol in detail
- Objective 3:** Discuss coordinator failure and recovery

A distributed transaction accesses data at multiple sites yet must remain atomic: either all sites commit or all abort. The challenge is that sites can fail independently and messages can be lost, so no site can decide alone. The transaction manager at each site coordinates with a global coordinator, and the protocol must guarantee atomicity despite crashes.

Two-phase commit (2PC) is the standard solution. Phase 1 (voting): the coordinator asks all participants to prepare; each participant force-writes its intentions to stable storage and replies YES (ready) or NO. Phase 2 (decision): if all voted YES, the coordinator sends COMMIT; otherwise ABORT. Participants acknowledge. The force-write before voting is the crux: a YES vote is a promise that the participant can commit even after crashing, which is what makes the protocol correct.

2PC's weakness is blocking: if the coordinator crashes after participants voted YES but before the decision arrives, participants block, holding locks indefinitely. Three-phase commit adds a pre-commit phase to avoid blocking but costs an extra round and still has edge cases; in practice, systems use 2PC with coordinator recovery via logging and backup coordinators. Very important: understand exactly what is force-written when, because recovery correctness arguments in exams hinge on it.

Key Definitions

Two-phase commit is defined as an atomic commitment protocol with a voting phase and a decision phase coordinated across sites.

Coordinator is defined as the site/process that drives a distributed transaction to a global commit or abort decision.

Blocking is defined as a state where participants cannot proceed because the coordinator's decision is unknown after a failure.

Key Points to Remember

- Distributed atomicity needs a protocol because sites fail independently.

- 2PC: phase 1 voting (prepare + force-write), phase 2 decision (commit/abort).
- Force-write before YES vote is what guarantees recoverability.
- Blocking on coordinator failure is 2PC's main weakness; 3PC mitigates at extra cost.

Unit Summary

Two-phase commit achieves distributed atomicity through voting and decision phases with careful logging; its blocking behavior under coordinator failure is the key limitation.

Exam Pointer

Very important: be able to walk through 2PC message by message, including a coordinator-crash scenario, and explain why the force-write before voting is essential.

Unit 6: Concurrency Control in Distributed Databases

Learning Objectives

- Objective 1:** Explain why centralized locking is insufficient
- Objective 2:** Describe distributed two-phase locking and timestamping
- Objective 3:** Discuss optimistic concurrency control

Concurrency control keeps concurrent transactions isolated. In a distributed database, lock managers are themselves distributed, so a transaction may wait on locks at several sites, and lock conflicts can span sites. Centralized locking would create a bottleneck and a single point of failure, so practical systems distribute lock management, typically with each site managing locks for its own data and transactions acquiring locks where the data lives.

Distributed two-phase locking extends the classic protocol: transactions acquire locks in a growing phase and release in a shrinking phase, with a distributed deadlock detector watching for cross-site wait cycles. Distributed timestamping instead orders transactions by unique global timestamps (often Lamport-style combining local time with site IDs); the Thomas write rule can even skip obsolete writes safely. Each method trades message overhead against abort rates.

Optimistic concurrency control lets transactions execute freely on private copies and validates at commit, aborting on conflict. It shines when conflicts are rare, as in many read-heavy distributed workloads, because it avoids locking messages entirely. The choice between pessimistic (locking) and optimistic methods depends on contention level, and exam questions often ask you to justify a choice for a given workload.

Key Definitions

Two-phase locking is defined as a protocol where a transaction acquires all locks before releasing any, guaranteeing serializability.

Timestamp ordering is defined as a concurrency method that orders conflicting operations by transaction timestamps.

Optimistic concurrency control is defined as a method that executes without locks and validates before commit, aborting on conflict.

Key Points to Remember

- Lock management must be distributed to avoid bottlenecks and single points of failure.
- Distributed 2PL needs cross-site deadlock detection; timestamping needs globally unique timestamps.
- Thomas write rule safely ignores obsolete writes in timestamp schemes.
- Optimistic methods suit low-conflict workloads by avoiding lock messages.

Unit Summary

Distributed concurrency control distributes lock management (2PL with deadlock detection), timestamp ordering, or optimistic validation, chosen according to contention and message costs.

Exam Pointer

Exam pointer: compare locking vs timestamping vs optimistic control for a stated workload, justifying message overhead and abort behavior.

Unit 7: Distributed Deadlock Detection and Recovery

Learning Objectives

- Objective 1:** Model deadlocks with wait-for graphs
- Objective 2:** Compare centralized, hierarchical and distributed detection
- Objective 3:** Explain recovery: victim selection and restart

Deadlock occurs when transactions wait on each other in a cycle. In distributed systems the wait-for graph is itself distributed: an edge at site A (T1 waits for T2) and an edge at site B (T2 waits for T1) form a global cycle invisible to either site alone. This is why local deadlock detectors are insufficient and global detection is a research-level problem in its own right.

Three detection architectures exist. Centralized detection ships all local wait-for information to one site, which is simple but creates communication overhead and a bottleneck. Hierarchical detection organizes sites in a tree, combining graphs upward. Fully distributed approaches (like the Chandy-Misra-Haas edge-chasing algorithm) send probes along wait edges; a probe returning to its origin proves a cycle. Edge-chasing detects cycles without global collection but needs careful handling of false positives from timing.

Prevention is an alternative: wait-die and wound-wait use timestamps to decide whether a requester waits or aborts, guaranteeing no cycles ever form. When detection finds a deadlock, recovery selects a victim (usually the youngest or cheapest-to-restart transaction) and rolls it back, releasing its locks. Very important: exams love asking you to construct a global wait-for graph from per-site edges and identify the cycle.

Key Definitions

Wait-for graph is defined as a directed graph where an edge $T_i \rightarrow T_j$ means T_i waits for a lock held by T_j ; a cycle means deadlock.

Edge-chasing is defined as a distributed deadlock detection algorithm that forwards probes along wait edges to find cycles.

Wait-die / wound-wait is defined as timestamp-based prevention schemes that abort one party to avoid cycles.

Key Points to Remember

- Global cycles can be invisible locally; per-site detectors are insufficient.
- Detection: centralized (simple, bottleneck), hierarchical, distributed edge-chasing.
- Prevention: wait-die and wound-wait use timestamps to avoid cycles entirely.
- Recovery: pick a victim (youngest/cheapest) and roll back.

Unit Summary

Distributed deadlocks need global detection (centralized, hierarchical, or edge-chasing) or timestamp-based prevention, with victim rollback as recovery.

Exam Pointer

Very important: practice building a global wait-for graph from site-local edges and finding the cycle. Also compare wait-die vs wound-wait behavior.

Unit 8: Object-Oriented and Object-Relational Databases

Learning Objectives

Objective 1: Explain OODBMS concepts: identity, encapsulation, inheritance

Objective 2: Describe the ODMG standard and OQL

Objective 3: Explain how object-relational systems extend SQL

Object-oriented DBMS grew from the mismatch between objects in programs and tables in databases. An OODBMS stores objects directly with three defining features: object identity (each object has a unique OID independent of its attribute values), encapsulation (state plus behavior stored together), and class hierarchies with inheritance. Complex objects like CAD designs or multimedia, painful to shred into tables, map naturally. Persistence becomes transparent: the program's objects simply survive.

The ODMG standard tried to unify the field with an object model, Object Definition Language (ODL) for schemas, and Object Query Language (OQL), a SQL-like language over objects supporting path expressions and method calls. OQL never displaced SQL, partly because the relational ecosystem was entrenched and partly because pure OODBMS lacked declarative query optimization maturity. Still, its ideas survive in object-relational mappers used everywhere today.

Object-relational DBMS took the pragmatic path: keep the relational engine and add object features. SQL:1999 introduced user-defined types, row types, reference types, and typed tables with inheritance hierarchies. PostgreSQL and Oracle implement these: you can define a type, create tables of that type, and query with both relational and object semantics. Very important: the exam contrast is philosophical (OODBMS: objects first) versus pragmatic (ORDBMS: relations first, objects added).

Key Definitions

Object identity (OID) is defined as a system-generated unique identifier for an object, independent of its state.

Encapsulation is defined as bundling data with the methods that operate on it.

Object-relational DBMS is defined as a relational system extended with user-defined types, inheritance and references.

Key Points to Remember

- OODBMS pillars: object identity, encapsulation, inheritance; natural for complex objects.
- ODMG gave ODL (schema) and OQL (query); OQL lost to SQL's ecosystem.
- ORDBMS (SQL:1999): user-defined types, typed tables, references; relations first.
- Modern ORMs carry OODBMS ideas into mainstream development.

Unit Summary

OODBMS stores objects natively with identity and inheritance; ODMG standardized OQL; object-relational systems pragmatically add types and inheritance to SQL engines.

Exam Pointer

Exam pointer: contrast OODBMS vs ORDBMS philosophically and give one example system or feature of each. OQL vs SQL comparison is standard.

Unit 9: Semi-Structured Data: XML Databases

Learning Objectives

Objective 1: Explain XML data modeling and schema languages

Objective 2: Describe XPath and XQuery

Objective 3: Discuss XML storage strategies

Semi-structured data has structure but no rigid schema: XML documents, JSON, web data. XML models data as ordered labeled trees with elements, attributes and text. Unlike tables, XML tolerates missing elements, repeated elements and mixed content, which suits document-centric and integration scenarios. Schema languages constrain this flexibility: DTD is the legacy simple form, while XML Schema adds data types, namespaces and richer constraints.

Querying XML needs tree-aware languages. XPath addresses nodes with path expressions (e.g. `//book[price<30]/title` selects titles of cheap books), supporting axes, predicates and functions. XQuery builds on XPath into a full FLWOR language (for-let-where-order-return) that can join, restructure and construct new XML, making it the XML analogue of SQL. Understanding FLWOR structure is the core skill of this unit.

Storage strategies split three ways: store XML as text (simple, slow queries), shred into relational tables (fast for data-centric regular XML, lossy for documents), or use native XML databases (eXist-db, BaseX) that index the tree structure directly. The choice depends on how regular the documents are and whether round-tripping fidelity matters. Very important: exams often ask when

native storage beats shredding; answer in terms of document irregularity and fidelity needs.

Key Definitions

Semi-structured data is defined as data with flexible, self-describing structure rather than a fixed schema.

XPath is defined as a language for addressing parts of an XML document via path expressions.

XQuery is defined as a query language for XML built on XPath with FLWOR expressions.

Key Points to Remember

- XML = ordered labeled trees; tolerates irregularity that tables cannot.
- DTD (simple) vs XML Schema (typed, richer); XPath addresses, XQuery (FLWOR) queries.
- Storage: text (simple), shredded to relations (regular data), native XML DB (documents).
- Native storage wins when documents are irregular and fidelity matters.

Unit Summary

XML models semi-structured data as trees; XPath/XQuery query it; storage ranges from text to shredded relations to native XML databases chosen by regularity and fidelity needs.

Exam Pointer

Very important: write sample XPath and XQuery FLWOR expressions; compare storage strategies for a given scenario.

Unit 10: Data Warehousing and OLAP

Learning Objectives

Objective 1: Define data warehouses and contrast with OLTP

Objective 2: Explain dimensional modeling: facts, dimensions, star schema

Objective 3: Describe OLAP operations and ETL

A data warehouse is a subject-oriented, integrated, time-variant, non-volatile collection of data supporting management decisions. It differs fundamentally from OLTP systems: OLTP handles many small concurrent transactions on current data, while warehouses handle few complex analytical queries over historical data. The warehouse is loaded periodically and never updated in place, which is what non-volatile means.

Dimensional modeling organizes warehouse data for analysis. Fact tables hold measurable events (sales amounts) with foreign keys to dimension tables (time, product, store) holding descriptive context. The star schema places one fact table at the center with denormalized dimensions radiating out; the snowflake schema normalizes dimensions further. Granularity, the level of detail in facts, is the most consequential design decision: too fine wastes space, too coarse loses analytical power.

Populating the warehouse is ETL: extract from sources, transform (clean, conform, deduplicate), load into the warehouse. OLAP then provides multidimensional analysis with operations: roll-up (aggregate up a hierarchy), drill-down (disaggregate), slice (fix one dimension), dice (sub-cube), and pivot (rotate

axes). Very important: be able to trace a business question to the OLAP operations that answer it.

Key Definitions

Data warehouse is defined as a non-volatile, integrated, historical data store built for decision support.

Star schema is defined as a dimensional design with a central fact table joined to denormalized dimension tables.

ETL is defined as extract, transform, load: the pipeline that populates a warehouse.

Key Points to Remember

- Warehouse vs OLTP: analytical/historical/few-complex vs transactional/current/many-small.
- Facts = measurable events; dimensions = descriptive context; granularity is the key design choice.
- Star (denormalized dims) vs snowflake (normalized dims).
- OLAP: roll-up, drill-down, slice, dice, pivot; fed by ETL.

Unit Summary

Data warehouses store integrated historical data modeled dimensionally (facts and dimensions, star schema), populated by ETL and analyzed with OLAP operations.

Exam Pointer

Very important: design a star schema for a given business (e.g. retail sales) and map business questions to OLAP operations. Warehouse-vs-OLTP contrast is guaranteed material.

Unit 11: Parallel Database Systems

Learning Objectives

Objective 1: Describe shared-memory, shared-disk and shared-nothing architectures

Objective 2: Explain partitioning strategies and their effects

Objective 3: Discuss parallel query execution and speedup/scaleup

Parallel databases speed queries by dividing work across processors. Three architectures dominate. Shared-memory (SMP) has all processors accessing common memory and disks: easy to program but limited scalability as the bus saturates. Shared-disk gives each processor private memory but common disks: better availability, though the interconnect to disks becomes the bottleneck. Shared-nothing gives each node private memory and disks, communicating by messages: hardest to program but scales to thousands of nodes, which is why it won for data warehousing.

Data placement drives performance. Partitioning strategies include round-robin (even spread, no locality), hash partitioning (good for point lookups and joins on the key), and range partitioning (good for range scans but risks skew). Skew, where one node gets disproportionate data or work, is the great enemy: the query finishes when the slowest node finishes. Choosing partition keys to avoid skew is a core DBA skill.

Parallelism comes in flavors: inter-query (different queries on different nodes), inter-operator (different operators concurrently), and intra-operator (one operator partitioned across nodes). Success is measured by speedup (same job, more nodes, less time) and scaleup (bigger job, more nodes, same time). Linear speedup is the ideal; skew and coordination overhead are what prevent it.

Key Definitions

Shared-nothing is defined as a parallel architecture where nodes share only the network, each with private memory and disks.

Partitioning is defined as distributing table data across nodes by round-robin, hash or range.

Skew is defined as uneven distribution of data or work that leaves some nodes idle while others bottleneck.

Key Points to Remember

- Architectures: shared-memory (easy, limited), shared-disk (available, interconnect bottleneck), shared-nothing (scales best).
- Partitioning: round-robin, hash, range; choose by query pattern; avoid skew.
- Parallelism: inter-query, inter-operator, intra-operator.
- Metrics: speedup and scaleup; linear speedup is the ideal.

Unit Summary

Parallel databases divide data and work across nodes; shared-nothing with smart partitioning scales best, and speedup/scaleup measure success against the enemy of skew.

Exam Pointer

Exam pointer: compare the three architectures on scalability and failure handling, and recommend a partitioning strategy for a given query mix.

Unit 12: NoSQL and NewSQL Systems

Learning Objectives

Objective 1: Explain the CAP theorem and BASE

Objective 2: Classify NoSQL stores: key-value, document, column, graph

Objective 3: Describe NewSQL's goals

NoSQL systems abandon the one-size-fits-all relational model for web-scale workloads. The CAP theorem frames their design: a distributed system can guarantee only two of consistency, availability and partition tolerance; since partitions happen, systems choose between consistency (CP) and availability (AP). BASE (Basically Available, Soft state, Eventually consistent) is the relaxed alternative to ACID that many NoSQL stores adopt, accepting temporary inconsistency for scale and uptime.

Four NoSQL families cover different data shapes. Key-value stores (Redis, DynamoDB) are ultra-fast lookup tables. Document stores (MongoDB, CouchDB) hold JSON-like documents with flexible schemas and rich queries. Wide-column stores (Cassandra, HBase) scale writes massively across clusters. Graph databases (Neo4j) model relationships as first-class citizens for traversals like

friend-of-friend or fraud rings. The data model, not marketing, should drive selection.

NewSQL systems (Google Spanner, CockroachDB) try to have it both ways: horizontal scalability with ACID guarantees, using innovations like TrueTime clock synchronization and Raft/Paxos consensus. Very important: the exam theme is trade-offs, not winners. For each system class, be able to state what it sacrifices and which workload justifies the sacrifice.

Key Definitions

CAP theorem is defined as a distributed store can provide at most two of consistency, availability and partition tolerance.

BASE is defined as Basically Available, Soft state, Eventually consistent: a relaxed consistency model for scalable systems.

Eventual consistency is defined as a guarantee that replicas converge given no new updates, without promising when.

Key Points to Remember

- CAP: pick two; partitions force a CP-vs-AP choice in practice.
- NoSQL families: key-value, document, wide-column, graph; match to data shape.
- BASE trades immediate consistency for availability and scale.
- NewSQL (Spanner, CockroachDB) aims for scale + ACID via consensus and synchronized clocks.

Unit Summary

NoSQL systems trade consistency for scale under CAP/BASE in four data-model families; NewSQL attempts to restore ACID at scale.

Exam Pointer

Very important: classify a given system or workload into CP/AP and a NoSQL family, justifying the choice. CAP definitions must be precise.

Unit 13: Advanced Topics: Security, Recovery, and Trends

Learning Objectives

Objective 1: Outline database security mechanisms

Objective 2: Explain distributed recovery with logging

Objective 3: Survey trends: cloud, in-memory, blockchain-adjacent ideas

Security in advanced systems layers several mechanisms. Authentication verifies users, authorization (discretionary or role-based access control) limits them to needed data, and views plus encryption protect data at rest and in transit. Inference control prevents attackers from combining allowed queries to deduce forbidden facts, a subtle problem in statistical and distributed databases. Auditing logs who accessed what, completing the accountability loop.

Recovery in distributed systems extends write-ahead logging across sites: each site logs its actions, and the 2PC protocol's force-writes double as recovery records. After a crash, a participant consults

its log: if it voted YES but never learned the decision, it must contact the coordinator or peers to resolve the transaction rather than unilaterally deciding. Checkpoints bound recovery time. The interplay of commit protocols and logging is exactly what makes distributed recovery examinable.

Current trends reshape the field continuously. Cloud-native databases separate compute from storage for elastic scaling. In-memory systems (SAP HANA, Redis) trade RAM cost for microsecond latency. Multi-model databases blend relational, document and graph in one engine. Very important: whatever the trend, evaluate it with the course's permanent questions: where is the data, what moves during queries, and how are ACID properties preserved.

Key Definitions

Role-based access control is defined as permissions assigned to roles, with users granted roles, simplifying administration.

Write-ahead logging is defined as recording intended changes in a log before applying them, enabling recovery.

Inference control is defined as preventing deduction of sensitive data from combinations of authorized queries.

Key Points to Remember

- Security layers: authentication, authorization (RBAC), views, encryption, auditing, inference control.
- Distributed recovery = per-site WAL + 2PC logs + coordinator inquiry after crashes.
- Never unilaterally decide a transaction in doubt; consult the coordinator.
- Evaluate every trend with: data placement, query data movement, ACID preservation.

Unit Summary

Advanced DBMS security layers access control with encryption and inference protection; distributed recovery combines logging with commit protocols; trends are judged by the course's core questions.

Exam Pointer

Exam pointer: walk through crash recovery of a 2PC participant that voted YES, and list security layers with one mechanism each.