Data Engineering Cheat Sheet

Son Hoang Pham

November 27, 2025

1 Principles in Database Analysis & Design

1.1 Design Phases Overview

1. **Conceptual:** *Goal/Focus* — capture requirements and semantics (implementation-independent); *Key Model/Tool* — ER/EER; *Output* — conceptual schema (entities, attributes, relationships, constraints).
2. **Logical:** *Goal/Focus* — map conceptual to target DBMS model (e.g., relational); *Key Model/Tool* — ER-to-relational mapping, normalization (FDs); *Output* — relational schema (tables, keys, integrity constraints).
3. **Physical:** *Goal/Focus* — specify storage structures and access paths for performance; *Key Model/Tool* — workload analysis, indexing, file organization, hashing; *Output* — internal schema (storage structures, indexes, access paths).

1.2 Conceptual Design Principles

- **Requirements Analysis:** Engage users/domain experts to capture data requirements and functional requirements (operations/transactions).
- **ER Components: Entity** (e.g., EMPLOYEE), **Attribute** (simple/composite/multivalued/derived), **Relationship** (associations among entities).
- **Structural Constraints:** Cardinality ratios (1:1, 1:N, M:N) and participation constraints (*total* vs *partial*).
- **Weak Entities:** Identified via an **identifying relationship** to an **owner** entity type and a **partial key**; weak entities have *total participation* in the identifying relationship.
- **Top-Down Refinement:** Iteratively refine broad entities; apply specialization/generalization (EER).

1.3 Logical Design Principles

1.3.1 Relational Model Fundamentals

- **Structure:** Relation schema $R(A_1, \ldots, A_n)$; tuples are unordered and duplicates are disallowed in the formal model.
- **Integrity Constraints:** Domain constraints (atomic, typed), key constraints (super/candidate/primary), entity integrity (primary key not NULL), referential integrity (foreign key values appear in referenced primary key).

1.3.2 Normalization Theory

- **Functional Dependency (FD):** $X \to Y$ means tuples equal on X must be equal on Y.
- **Design Properties:**
  1. **Lossless Join (Non-Additive):** Decompositions must not create spurious tuples.
  2. **Dependency Preservation:** Original FDs enforceable on decomposed relations.
  3. **Minimal Redundancy:** Avoid update anomalies (insertion, deletion, modification).
- **Normal Forms:** 1NF (atomic), 2NF (full FD on key), 3NF (no transitive FD), BCNF (for every FD $X \to A$, X is a superkey; may not preserve dependencies). 4NF/5NF address multivalued and join dependencies.
- **Denormalization:** Store joins as base relations to improve performance at the cost of anomalies.

1.4 Physical Design Principles

- **Storage Architecture:** Persistent data on disks/SSDs in fixed-size **blocks**.
- **Workload Analysis (Job Mix):** Identify accessed relations/files, selection conditions (equality/inequality/range), and update vs retrieval frequencies.
- **Indexing Structures:** Ordered indices (B+-Trees) and hash indices; **primary/clustering** index (determines physical order; at most one per file) vs **secondary** indices.
- **Query Optimization:** Cost-based (statistics-driven) and heuristic rules (push selections/projections early) to choose efficient plans.

2 Data Storage & Indexing

2.1 Index Fundamentals

2.1.1 What is an Index?

- **Purpose:** Accelerate data retrieval by creating auxiliary access paths to records.
- **Search Key:** Attribute (s) used to locate records; need not be the primary key; can be composite (multi-column).
- **Trade-off:** Faster reads vs slower writes (index maintenance overhead on INSERT/UPDATE/DELETE).

2.1.2 Index Classification

- **By Structure:**
  - *Ordered:* Entries sorted (e.g., B+-tree); supports range queries.
  - *Hash:* Key hashed to bucket; fast equality lookups only.
- **By Density:**
  - *Dense:* One entry per distinct search-key value.
  - *Sparse:* One entry per block (or per distinct clustering value); smaller, cheaper to maintain.
- **By Physical Ordering:**
  - *Primary/Clustering Index:* Search key determines file's physical sort order (at most one per table); typically sparse.
  - *Secondary Index:* Alternative access path independent of physical order; typically dense or uses indirection for non-unique keys.

2.2 B-Trees & B+-Trees

2.2.1 Why B+-Trees?

- **Goal:** Minimize expensive disk I/O by keeping tree height small via high fan-out.
- **Structure:** Balanced tree; internal nodes store keys + child pointers; leaf nodes store keys + data pointers (or record IDs) + sibling link.
- **Operations:** Search/insert/delete in $O(\log_{fo} n)$ block accesses; splits/merges maintain balance.
- **B+- vs B-Tree:** B+- stores data pointers only in leaves $\Rightarrow$ higher internal fan-out, efficient sequential scans via leaf chain.

2.2.2 Capacity Calculation Example

Given parameters: Block size $B = 512$ bytes; key size $V = 9$ bytes; data pointer $\text{Pr} = 7$ bytes; tree pointer $P = 6$ bytes.

Step 1: Calculate Order (Maximum Pointers per Node)

– **B-Tree Internal Node:** Holds p tree pointers + $(p-1)$ keys + $(p-1)$ data pointers.

$$(p \times 6) + ((p-1) \times (7+9)) \leq 512$$

$$6p + 16p - 16 \leq 512 \quad \Rightarrow \quad 22p \leq 528 \quad \Rightarrow \quad p = 23$$

– **B+-Tree Internal Node:** Holds p tree pointers + $(p-1)$ keys (no data pointers).

$$(p \times 6) + ((p-1) \times 9) \leq 512$$

$$6p + 9p - 9 \leq 512 \quad \Rightarrow \quad 15p \leq 521 \quad \Rightarrow \quad p = 34$$

– **B+-Tree Leaf Node:** Holds p_{leaf} key/data-pointer pairs + 1 next-pointer.

$$(p_{leaf} \times (7+9)) + 6 \leq 512$$

$$16 \times p_{leaf} \leq 506 \quad \Rightarrow \quad p_{leaf} = 31$$

Step 2: Estimate Total Capacity (69% Full)

– **B-Tree (3 levels):** Avg fan-out $fo = 23 \times 0.69 \approx 16$.
  – Level 0 (root): 15 entries, 16 pointers
  – Level 1: $16 \times 15 = 240$ entries
  – Level 2: $256 \times 15 = 3{,}840$ entries
  – **Total:** $15 + 240 + 3{,}840 \approx \mathbf{4{,}095}$ entries
– **B+-Tree (3 levels):** Internal $fo = 34 \times 0.69 \approx 23$; Leaf capacity $= 31 \times 0.69 \approx 21$.
  – Level 0: 22 entries, 23 pointers
  – Level 1: $23 \times 22 = 506$ entries, 529 pointers
  – Leaf level: $12{,}167 \times 21 \approx \mathbf{255{,}507}$ data pointers
– **Key Insight:** B+- holds $\sim 4\times$ more entries at same height due to lighter internal nodes.

2.2.3 Advanced Index Types

– **Composite Index:** Multi-column key (e.g., (City, LastName)); supports leftmost prefix queries (City), (City, LastName); order columns by selectivity.
– **Function-Based Index:** Index on expression (e.g., `LOWER(email)`); query must use same function to benefit.

2.3 Hash & Bitmap Indexes

2.3.1 Hash Indexes

– **Use case:** Fast equality lookups (point queries); no range support.
– **Collision handling:** Chaining with overflow buckets.
– **Dynamic variants:** Extendible/linear hashing grow incrementally without full rebuild.

2.3.2 Bitmap Indexes

Optimized for low-cardinality attributes (few distinct values).

– **Structure:** Sequential record numbering (0, 1, 2, ...); one bitmap per distinct value; bit $i = 1$ if record i has that value.
– **Example (5-row table):**
  – `gender='m'`: 10010 `gender='f'`: 01101
  – `income='L1'`: 11000 `income='L2'`: 00100
– **Query: gender='f' AND income='L2'**

$$01101 \text{ AND } 00100 = 00100 \quad \Rightarrow \quad \text{record 2}$$

– **Advantages:** Compact (1M rows = 125 KB per bitmap); fast bitwise ops; efficient for multi-condition filters; supports COUNT via bit-counting.

2.4 Query Optimization & Cost Analysis

2.4.1 Measures of Query Cost & Catalog Information

– **Primary metric (I/O):** Minimize block transfers (b) and random I/O accesses (seeks S). Time model: $b \times t_T + S \times t_S$.
– **Relation size:** r = tuples, b = file blocks.
– **Indexing details:** x = multilevel index height (e.g., B+-tree), b_{I1} = first-level index blocks.
– **Blocking factor:** bfr = tuples per block.
– **Selection cardinality:** $s = sl \times r$ where sl is selectivity fraction.
– **Distinct counts:** $NDV(A)$ = distinct values of attribute A.

2.4.2 Cost Functions for Selection

Selection (σ) can use file scans or index/hash access depending on available paths. Costs exclude write-out of the final result.

S1: Linear Search (Brute Force / A1)

– Worst case/non-key: $\mathbf{C_{S1a} = b}$.
– Average equality on a key: $\mathbf{C_{S1b} = b/2}$ (stop when found).

S2: Binary Search (ordered file)

$\mathbf{C_{S2} = \log_2 b + \lceil s/bfr \rceil - 1}$.

S3a: Primary Index (single record)

$\mathbf{C_{S3a} = x + 1}$.

S3b: Hash Key (single record)

$\mathbf{C_{S3b} = 1}$ (static/linear) or **2** (extendible).

S5: Clustering Index (equality on non-key / A3)

$\mathbf{C_{S5} = x + \lceil s/bfr \rceil}$.

S6a: Secondary Index (equality on non-key / A4)

$\mathbf{C_{S6a} = x + 1 + s}$ (worst case, scattered records).

S6b: Secondary Index (range query)

$\mathbf{C_{S6b} = x + (b_{I1}/2) + (r/2)}$.

Note: Time cost often modeled as $b \times t_T + S \times t_S$, separating transfer vs seek.

2.4.3 Worked Example: Selection on EMPLOYEE

Scenario: EMPLOYEE with $r_E = 10{,}000$, $b_E = 2{,}000$, $bfr_E = 5$. Available indices/access paths:

- **Salary** (clustering, non-key): $x = 3$, $s_{Salary} = 20$.
- **Ssn** (secondary, key): $x = 4$, $s_{Ssn} = 1$.
- **Dno** (secondary, non-key): $x = 2$, $s_{Dno} = 80$ (from 10,000/125).
- **Sex** (secondary, non-key): $x = 1$, $s_{Sex} = 5{,}000$ (from 10,000/2).

OP1: Equality on Key

Query: $\sigma_{\text{Ssn}='123456789'}(\text{EMPLOYEE})$.
– S1b (linear avg): $C_{S1b} = b_E/2 = 1{,}000$.
– S6a (secondary index on key): $C_{S6a} = x_{Ssn} + 1 = 4 + 1 = \mathbf{5}$.
– **Decision:** Choose S6a ($5 \ll 1{,}000$).

OP3: Equality on Non-Key

Query: $\sigma_{\text{Dno}=5}(\text{EMPLOYEE})$.
– S1a (linear): $C_{S1a} = b_E = \mathbf{2{,}000}$.
– S6a (secondary on Dno): $C_{S6a} = x_{Dno} + s_{Dno} = 2 + 80 = \mathbf{82}$.
– **Decision:** Choose S6a ($82 \ll 2{,}000$). If clustering on Dno existed: $3 + \lceil 80/5 \rceil = 19$ blocks.

OP4: Conjunctive Selection

Query: $\sigma_{\text{Dno}=5 \wedge \text{Salary}>30\,000 \wedge \text{Sex}='F'}(\text{EMPLOYEE})$. Optimizer compares access paths to get the initial candidate set, then checks remaining predicates in memory.
– Via Dno (S6a): $C = x_{Dno} + s_{Dno} = \mathbf{82}$.
– Via Salary range (clustered): $C \approx x_{Salary} + (b_E/2) = 3 + 1{,}000 = \mathbf{1{,}003}$.
– Via Sex (S6a): $C = x_{Sex} + s_{Sex} = 1 + 5{,}000 = \mathbf{5{,}001}$.
– Brute force (S1a): $C = 2{,}000$.
– **Decision:** Use Dno index (82), retrieve 80 tuples, then filter Salary $> 30\,000$ and Sex $=' F'$ in RAM.

2.4.4 Join Algorithms & Cost Comparison

Key Parameters

– **Join selectivity:** $js = |R \bowtie S|/(|R|\,|S|)$; for equi-join $js \approx 1/\max(NDV(A), NDV(B))$.
– **Join cardinality:** $jc = js\,|R|\,|S|$; **write cost**: jc/bfr_{result} blocks.
– **Buffers:** n_B = available buffer pages (impacts nested-loop cost).

Unified Scenario

EMPLOYEE ($|E| = 10{,}000$, $b_E = 2{,}000$) $\bowtie_{Dno=Dnumber}$
DEPARTMENT ($|D| = 125$, $b_D = 13$).
Index on $E.Dno$ (secondary: $x = 2$, $s = 80$).
$Dnumber$ primary key ($x = 1$).
Assume $js = 1/125$, $jc = 10{,}000$, $bfr_{result} = 4$ (write cost = 2,500 blocks), $n_B = 3$.

J1: Block Nested-Loop

Cost: $C_{J1} = b_R + \left\lceil \frac{b_R}{n_B - 2} \right\rceil b_S + \frac{jc}{bfr_{result}}$. Using DEPARTMENT as outer: $C_{J1} = 13 + \lceil 13/1 \rceil \times 2{,}000 + 2{,}500 = \mathbf{28{,}513}$.

J2: Indexed Nested-Loop

- **DEPARTMENT outer → EMPLOYEE inner:** Per lookup $= x + s = 2 + 80 = 82$. Total $= 13 + 125 \times 82 + 2{,}500 =$ **12,763**.
- **EMPLOYEE outer → DEPARTMENT inner:** Per lookup $= x + 1 = 1 + 1 = 2$. Total $= 2{,}000 + 10{,}000 \times 2 + 2{,}500 =$ **24,500**.

J3: Sort-Merge

If pre-sorted:

$$C_{J3} = b_E + b_D + \frac{jc}{bfr_{result}} = 2{,}000 + 13 + 2{,}500 = \mathbf{4{,}513}$$

. If not, add external sort cost per relation.

J4: Partition-Hash

Approximate cost: $C_{J4} \approx 3\,(b_E + b_D) + \frac{jc}{bfr_{result}} = 3 \times (2{,}000 + 13) + 2{,}500 =$ **8,539**.

Plan Decision

J3 (if sorted) < J4 < J2 (D outer) < J2 (E outer) < J1. Prefer hash join when unsorted; prefer sort-merge when sorted order exists.

2.5 Query Optimization Pipeline

2.5.1 Process Overview

1. **Parse & Validate:** Check syntax, schema compliance.
2. **Translate:** Convert SQL to relational algebra (query tree).
3. **Heuristic Optimization:** Apply transformation rules.
4. **Cost-Based Optimization:** Enumerate plans, estimate costs, pick minimum.
5. **Execute:** Materialize intermediates or pipeline results.

2.5.2 Heuristic Rules

- **Push selections (σ) down:** Filter early to reduce intermediate size.
- **Push projections (Π) down:** Reduce tuple width early.
- **Replace σ + Cartesian product ($\times$) with join ($\bowtie$):** Avoid expensive cross products.
- **Reorder joins:** Use commutativity/associativity to seek low-cardinality intermediates.
- **Left-deep trees:** Right child always a base table ⇒ enables index lookups, reduces search space.

2.5.3 Cost-Based Decisions

- **Access path selection:** Compare full scan, clustering index, secondary index, bitmap for each predicate.
- **Join algorithm:** Nested-loop (index/block), hash join, sort-merge based on cardinality, memory.
- **Join order:** Estimate join selectivity $js \approx 1/\max(NDV(A), NDV(B))$; cardinality $jc = js \times |R| \times |S|$.

2.6 Specialized Techniques

2.6.1 Write-Optimized Structures

- **LSM-Tree:** In-memory buffer (memtable) + sorted on-disk levels; sequential writes; periodic compaction; Bloom filters skip levels.
- **Buffer Tree:** B-tree variant with per-node write buffers; batches mutations down tree; better read latency than LSM.

2.6.2 Spatial & Multi-Key Access

- **Multi-key:** Composite index for prefix queries; covering index avoids table lookup.
- **Spatial:** R-tree (bounding rectangles), kd-tree, quadtree for geospatial/interval data; support range + nearest-neighbor queries.

3 Data Storage

3.1 Concepts

- **NoSQL motivation:** Horizontal scalability, availability, flexible schemas (BASE/CAP) vs strict ACID.
- **Big Data 3V's:** Volume (TB–PB–EB), Velocity (real-time), Variety (structured/semi/unstructured); often add Veracity and Value.
- **Schema strategy:** *Schema-on-write* (DW) vs *schema-on-read* (DL).
- **Distributed storage:** HDFS-like systems provide fault-tolerant, scalable storage across clusters.

3.1.1 NoSQL Motivation (BASE/CAP vs ACID)

- **ACID vs BASE:** RDBMS enforce ACID; NoSQL often adopt BASE (Basically Available, Soft state, Eventually consistent) to scale.
- **CAP trade-off:** In replicated distributed systems, cannot have Consistency, Availability, and Partition tolerance simultaneously. NoSQL commonly choose AP over strict C.
- **Eventual consistency:** Replicas may diverge transiently but converge if no further updates; acceptable for many web-scale apps.
- **Scale-out:** Horizontal scalability via sharding/partitioning and replication across commodity nodes.
- **Flexible schemas:** Self-describing records (JSON/BSON), semi-structured data, heterogeneous attributes per item.
- **Models:** Document (MongoDB), Key-Value (Redis/DynamoDB), Wide-column (BigTable/HBase/Cassandra), Graph (Neo4j).

3.1.2 Big Data Five V's

- **Volume:** Scale from TB to PB to EB; driven by logs, mobile, transactions, IoT sensors/RFID; requires massive parallelism.
- **Velocity:** High ingest rates and real-time/stream processing for fraud intrusion detection, monitoring.
- **Variety:** Structured, semi-structured, unstructured (web, social, location, images, video, logs, signals). Unstructured data is the major challenge.
- **Veracity:** Data credibility/quality varies; requires testing and suitability checks before analytics.
- **Value:** Analytics (descriptive/predictive/prescriptive) to derive business benefit and innovation.

3.1.3 Distributed Storage (HDFS-like)

- **Shared-nothing:** Each node has its own CPU/RAM/disk; coordination over network; scales economically with redundancy.
- **HDFS architecture:** Master NameNode manages namespace/metadata; DataNodes store blocks and serve I/O; clusters can have thousands of DataNodes.
- **Replication:** Blocks replicated (typically 3x) for durability and availability; clients read from nearest replica to maximize bandwidth.
- **Parallel I/O:** Many machines read/write concurrently, increasing aggregate throughput.
- **Append-only:** Simple coherency model optimized for batch: files are append-only (no random updates).
- **Ecosystem:** HDFS underpins MapReduce, HBase, and other big data tools.

3.2 Technologies

Document Store (MongoDB)

- Flexible *schema-on-read* for Variety; heterogeneous attributes per document.
- JSON/BSON documents (arrays, nested objects); denormalized designs for data locality.
- CRUD via `find(<cond>)`; auto-indexed _id for key-based retrieval.
- High availability via replica sets (primary/secondary reads).
- Horizontal scaling with sharding on a shard key (range/hash; query router).
- Read-only MapReduce over large collections.

Key-Value Cache (Redis)

- In-memory hash-map for very fast reads.
- Complex data structures (sets, queues) supported natively.
- Persistence via append-only log and snapshots for durability.
- Read-through cache pattern; application manages invalidation.
- Master-slave replication for high availability.

Graph DB (Neo4j)

- Property graph model: nodes, relationships, labels, properties.
- Efficient path queries; variable-length traversals via Cypher.
- Declarative language with arrow notation for patterns.
- Enterprise features: caching, clustering, master-slave replication.
- Centralized design optimized for graph workloads.

Wide-Column Stores (BigTable/HBase/Cassandra)

- Sparse multidimensional sorted maps: row key, column info, versions.
- LSM-tree storage converts random writes to sequential I/O.
- HBase on HDFS; ZooKeeper for coordination; regions by key range.
- Column families group storage; qualifiers defined dynamically.
- Cassandra: Dynamo-style leaderless replication; consistent hashing.
- Compaction strategies (e.g., time-window); SASI secondary indexes.

Data Warehouse Engines

- HiveQL on Hadoop; compiles to MapReduce/Tez/Spark execution plans.
- SerDe exposes raw files as tables; predicate pushdown.
- Columnar formats ORC/Parquet minimize I/O for analytics.
- Cloud OLAP: Snowflake/BigQuery/Redshift for interactive analytics.
- Parallel execution with scalable storage.

Processing Frameworks

- MapReduce: map/shuffle/reduce over HDFS/HBase; resilient via re-execution.
- Spark: in-memory RDDs; unified engines (SQL, graphs, ML, streaming).
- Spark on YARN; reads from HDFS/HBase.
- Tez: DAG execution; pipelines stages to avoid HDFS materialization.
- Foundation for higher-level systems like Hive.

3.3 Streaming

- **Purpose:** Unbounded, real-time processing for fraud/intrusion detection, tracking, monitoring, traffic estimation.

Messaging: Kafka Durable Log & CDC

- **Log storage:** Append-only disks; producers append, consumers read sequentially.
- **Partitioning/ordering:** Topic partitions with total order per partition via offsets.
- **Durability/replay:** Retention keeps tuples; consumption is read-only; consumers can replay from offsets.
- **CDC transport:** Change Data Capture streams preserve order; source DB is leader, followers rebuild state.
- **Log compaction:** Retains last write per key, enabling up-to-date snapshots of tables.

Compute: Flink/Spark & Cloud Analogs

- **Apache Flink:** True streaming engine; pipelined execution; checkpointing for recovery.
- **Spark Streaming:** Microbatching ($\tilde{1}$s) over RDDs; recompute on failure.
- **Cloud:** Kinesis (log-based streams), Dataflow (checkpointed pipelines), Azure Stream Analytics (managed streaming SQL).

Windows: Tumbling vs Hopping

- **Tumbling:** Fixed-length, adjacent, non-overlapping windows (e.g., 1 minute).
- **Hopping:** Fixed-length, overlapping windows (e.g., 5-minute window hopping every 1 minute).
- **Time semantics:** Event time vs processing time; manage late/straggler events.

Stream Joins: Time-Bounded & Stateful

- **Stream-Stream:** Windowed join on key with time bound (e.g., within 30 minutes).
- **Stream-Table:** Enrichment of events using a relation/changelog treated as a table.
- **Table-Table:** Joining changelogs to maintain materialized views (e.g., tweets $\times$ follows).
- **State:** Processor maintains keyed state and window buffers to match arrivals.

Exactly-Once Semantics

- **Goal:** Output equivalent to failure-free execution (no loss/duplicates).
- **Framework recovery:** Microbatching (Spark) & checkpointing (Flink) restart from consistent points; discard partial outputs.
- **Distributed transactions:** Atomic commits for state + messaging within the processor.
- **External idempotence:** Use unique, persistent offsets/keys to make side effects idempotent (dedupe on write).

3.4 Pipelines: ETL vs ELT

- **ETL:** Extract $\rightarrow$ Transform $\rightarrow$ Load (transform before landing).
- **ELT:** Extract $\rightarrow$ Load $\rightarrow$ Transform (use warehouse compute; schema-on-read).
- **Kafka:** Transport/CDC stream for Extract/Load.
- **Airflow:** Orchestrates batch dependencies for Transform/Load.

3.5 IoT

- **Sources:** Sensors/RFID (high-velocity, varied signal data).
- **Apps:** Smart farming, athlete monitoring, mobility tracking.
- **Protocols:** MQTT for real-time ingestion.

3.6 Comparisons

MongoDB vs Redis vs Neo4j

- **MongoDB:** Document store, BSON, replica sets, sharding, MapReduce.
- **Redis:** In-memory key-value/cache, persistence, replication; fast reads.
- **Neo4j:** Graph model, Cypher for paths, clustering; efficient graph queries.

Cassandra vs Hive vs Snowflake

- **Cassandra:** Wide-column, leaderless replication, eventual consistency, LSM + compaction, SASI.
- **Hive:** SQL-on-Hadoop compiled to MapReduce/Tez/Spark; SerDe for ORC/Parquet.
- **Snowflake:** Cloud DW OLAP; peers include BigQuery/Redshift.

BigTable vs BigQuery

- **BigTable:** Distributed wide-column storage; sparse sorted map; range queries; inspired HBase.
- **BigQuery:** Interactive OLAP analytics; Dremel lineage; SQL engines at web scale.

3.7 Data Warehousing

- **Characteristics:** Subject-oriented, integrated, non-volatile, time-variant; fewer queries but large scans.
- **Schemas:** Fact (measures + dimension keys) and dimension tables; Star vs Snowflake.
- **Storage:** Column-oriented favored; Teradata, Sybase IQ, Redshift; Oracle/HANA/SQL Server support columnar.
- **Big Data integration:** Hadoop with Hive/Spark SQL for SQL over distributed files.

3.8 Multi-Dimensional Models

- **Tabular (Dimensional):** Star/Snowflake schemas enable dimensional analysis.
- **Data Cube:** `GROUP BY CUBE` (all subsets); `ROLLUP` (hierarchical subsets).

3.9 DW vs DL vs Lakehouse

- **DW:** Schema-on-write; strong consistency; structured data; OLAP (ETL).
- **DL:** Schema-on-read; raw multi-format data; cheap storage; Hadoop/Spark for querying.
- **Lakehouse:** Inferred hybrid combining lake flexibility with warehouse management (ACID, schema enforcement, indexing); not explicitly defined in sources.