Tổng hợp Kiến thức Môn Kỹ thuật Dữ liệu (Data Engineering)

Học viên Cao học KHMT Bách Khoa TP.HCM (HCMUT)

Ngày 31 tháng 12 năm 2025

1 Nguyên lý Phân tích & Thiết kế CSDL

1.1 Tổng quan Các giai đoạn Thiết kế

1. **Mức Quan niệm (Conceptual):** *Mục tiêu* — nắm bắt yêu cầu và ngữ nghĩa (độc lập với cài đặt); *Mô hình/Công cụ* — ER/EER; *Đầu ra* — lược đồ quan niệm (thực thể, thuộc tính, mối kết hợp, ràng buộc).
2. **Mức Logic:** *Mục tiêu* — ánh xạ từ mức quan niệm sang mô hình DBMS đích (ví dụ: quan hệ); *Mô hình/Công cụ* — ánh xạ ER-sang-quan hệ, chuẩn hóa (FDs); *Đầu ra* — lược đồ quan hệ (bảng, khóa, ràng buộc toàn vẹn).
3. **Mức Vật lý:** *Mục tiêu* — xác định cấu trúc lưu trữ và đường dẫn truy xuất để tối ưu hiệu năng; *Mô hình/Công cụ* — phân tích tải, chỉ mục, tổ chức tập tin, băm; *Đầu ra* — lược đồ trong (cấu trúc lưu trữ, chỉ mục, đường dẫn truy xuất).

1.2 Nguyên lý Thiết kế Mức Quan niệm

- **Phân tích Yêu cầu:** Làm việc với người dùng/chuyên gia nghiệp vụ để nắm bắt yêu cầu dữ liệu và yêu cầu chức năng (thao tác/giao dịch).
- **Thành phần ER: Thực thể** (vd: NhanVien), **Thuộc tính** (đơn/phức hợp/đa trị/dẫn xuất), **Mối kết hợp** (sự liên kết giữa các thực thể).
- **Ràng buộc Cấu trúc:** Tỷ số bản số (1:1, 1:N, M:N) và ràng buộc tham gia (*toàn phần* vs *từng phần*).
- **Thực thể Yếu:** Được xác định thông qua **mối kết hợp xác định** với một thực thể **chủ** và một **khóa bộ phận**; thực thể yếu tham gia *toàn phần* vào mối kết hợp xác định.
- **Tinh chỉnh Top-Down:** Tinh chỉnh lặp lại các thực thể tổng quát; áp dụng chuyên biệt hóa/tổng quát hóa (EER).

1.3 Nguyên lý Thiết kế Mức Logic

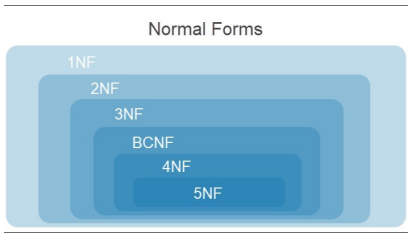
1.3.1 Cơ bản về Mô hình Quan hệ

- **Cấu trúc:** Lược đồ quan hệ $R(A_1, \ldots, A_n)$; các bộ (tuple) không có thứ tự và không cho phép trùng lặp trong mô hình hình thức.
- **Ràng buộc Toàn vẹn:** Ràng buộc miền giá trị (nguyên tố, có kiểu), ràng buộc khóa (siêu khóa/khóa ứng viên/khóa chính), toàn vẹn thực thể (khóa chính không NULL), toàn vẹn tham chiếu (giá trị khóa ngoại phải xuất hiện trong khóa chính được tham chiếu).

1.3.2 Lý thuyết Chuẩn hóa (Normalization Theory)

- **Mục tiêu:** Giảm thiểu dư thừa, tránh dị thường (Thêm, Xóa, Sửa).
- **Phụ thuộc Hàm (FD):** $X \to Y$ (Nếu $t_1[X] = t_2[X]$ thì $t_1[Y] = t_2[Y]$).
- **Các loại Key:**
  - *Superkey:* Xác định duy nhất một bộ.
  - *Candidate Key:* Superkey tối thiểu.
  - *Prime Attribute:* Thuộc tính nằm trong bất kỳ Candidate Key nào.

Các Dạng Chuẩn (Normal Forms)



(Hierarchy: BCNF ⊂ 3NF ⊂ 2NF ⊂ 1NF)

1NF (Atomic): Miền giá trị nguyên tố.
↪ *Vi phạm:* Thuộc tính đa trị, lồng nhau, lặp lại nhóm.
2NF (No Partial): Là 1NF + Thuộc tính *non-prime* phụ thuộc đầy đủ vào khóa.
↪ *Vi phạm:* $\exists X \subsetneq Key$ sao cho $X \to NonPrime$.
(Chỉ xảy ra nếu Key là khóa phức hợp).
3NF (No Transitive): Là 2NF + Không có phụ thuộc bắc cầu giữa các *non-prime*.
→ *Định nghĩa:* Với mọi $X \to A$ (không tầm thường), phải thỏa: (a) X là Superkey **HOẶC** (b) A là Prime Attribute.
BCNF (Strict): Nghiêm ngặt hơn 3NF.
→ *Định nghĩa:* Với mọi $X \to A$, X **bắt buộc là Superkey**.
(Khác biệt: BCNF không chấp nhận ngoại lệ "A là Prime"như 3NF).

Tính chất Phân rã (Decomposition Properties)

1. **Kết nối bảo toàn thông tin (Lossless Join):** *(Bắt buộc)*
   Để phân rã R thành R_1, R_2 không bị mất dữ liệu, điều kiện là:
   $(R_1 \cap R_2) \to R_1$ **HOẶC** $(R_1 \cap R_2) \to R_2$.
   (Giao của 2 bảng phải là khóa của ít nhất 1 bảng).
2. **Bảo toàn phụ thuộc (Dependency Preservation):**
   Các FD ban đầu có thể được kiểm tra riêng lẻ trên từng R_i mà không cần join lại. (BCNF có thể không bảo toàn phụ thuộc).

1.3.3 Phi chuẩn hóa (Denormalization)

- **Mục tiêu:** Cải thiện hiệu suất đọc bằng cách đưa dư thừa vào lược đồ, ngược với chuẩn hóa.
- **Động lực:** Tránh Join tốn kém; giảm độ phức tạp truy vấn; tăng locality dữ liệu.

Kỹ thuật Denormalization:

- **Materialized Views:** Kết quả truy vấn được tính trước và lưu trữ; cập nhật khi dữ liệu thay đổi.
- **Precomputed Aggregates:** Lưu giá trị tổng hợp (COUNT, SUM) trong bản ghi để tránh tính lại.
  VD: Lưu số email chưa đọc trong bảng User thay vì đếm mỗi lần.
- **Document Databases:** Nhúng (embedding) dữ liệu liên quan trong một document thay vì tham chiếu.
  VD: MongoDB nhúng thông tin Worker vào document Project.
- **Star Schema (DW):** Dimension tables được denormalize (VD: Brand, Category trong dim_product).
- **Microservices:** Sao chép (replicate) dữ liệu giữa các service để tách biệt và giảm phụ thuộc.

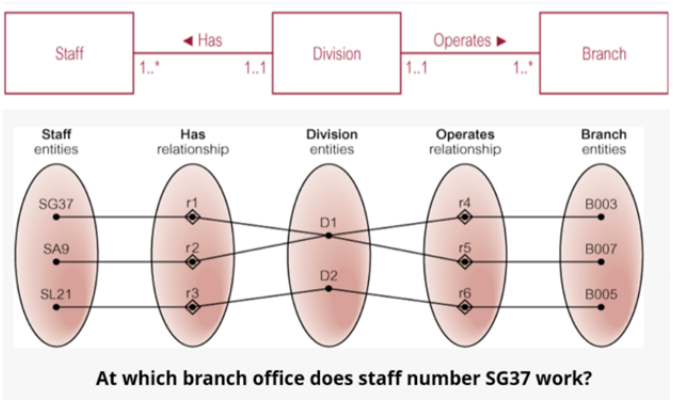
Đánh đổi (Trade-offs):

- **Write Overhead:** Mỗi cập nhật phải sửa tất cả bản sao dư thừa ⇒ ghi chậm hơn, phức tạp hơn.
- **Data Inconsistency:** Rủi ro không nhất quán nếu một bản sao được cập nhật mà bản khác thì không.
- **Storage Cost:** Lưu trữ dữ liệu trùng lặp tốn bộ nhớ hơn.

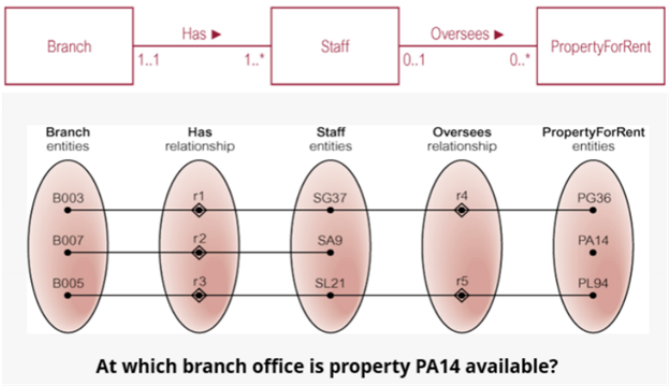
Khi nào dùng: OLAP/DW (đọc nhiều), NoSQL (thiếu join), microservices (tách biệt). *Tránh:* OLTP cần ACID chặt.

1.3.4 Bẫy Thiết kế CSDL (Design Traps)

– **Bẫy Kết nối trong ER:**



(Fan Trap)



(Chasm Trap)

  – *Fan Trap:* Đường dẫn giữa các thực thể mơ hồ do nhiều quan hệ 1:N phân nhánh từ một thực thể. VD: NhanVien → PhongBan → ChiNhanh (không xác định được nhân viên làm ở chi nhánh nào). *Giải pháp:* Thêm quan hệ trực tiếp NhanVien-ChiNhanh.
  – *Chasm Trap:* Đường dẫn không tồn tại do tham gia tùy chọn. VD: KhanhHang → TaiSan → ChiNhanh (nếu tài sản chưa niêm yết thì không liên kết được khách hàng với chi nhánh). *Giải pháp:* Đổi sang tham gia bắt buộc hoặc thêm quan hệ trực tiếp.

– **Dị thường Cập nhật (Update Anomalies):** Do thiết kế không chuẩn hóa:
  – *Insertion:* Không thể thêm phòng ban mới nếu chưa có nhân viên.
  – *Deletion:* Xóa nhân viên cuối cùng làm mất thông tin phòng ban.
  – *Modification:* Thay đổi tên phòng ban phải cập nhật nhiều bộ.

– **Bộ giả (Spurious Tuples):** Kết nối các quan hệ phân rã sai (không qua PK/FK hợp lệ) tạo ra bản ghi ảo. *Giải pháp:* Dùng phân rã bảo toàn thông tin (lossless join).

– **Bẫy NULL:** Quá nhiều thuộc tính NULL lãng phí bộ nhớ và gây khó khăn trong truy vấn tổng hợp.

– **Sai lầm phổ biến:** Dùng PK của thực thể này làm thuộc tính của thực thể khác thay vì mô hình hóa quan hệ; gán PK vào thuộc tính của quan hệ; dùng thuộc tính đơn trị khi cần đa trị.

– **Entity Trap (Kiến trúc):** Thiết kế component 1-1 với bảng DB (VD: CustomerManager cho bảng Customer) thay vì theo workflow nghiệp vụ → vi phạm tách biệt dữ liệu trong microservices.

Thực hành tốt: Dùng BCNF/3NF; phân rã bảo toàn thông tin; đảm bảo đường dẫn ER rõ ràng; thiết kế theo hành vi nghiệp vụ, không theo thực thể.

1.4 Nguyên lý Thiết kế Mức Vật lý

– **Kiến trúc Lưu trữ:** Dữ liệu bền vững trên đĩa/SSD trong các **khối (block)** kích thước cố định.

– **Phân tích Tải (Job Mix):** Xác định các quan hệ/tập tin thường truy cập, điều kiện chọn (bằng/khác/khoảng), và tần suất cập nhật so với truy vấn.

– **Cấu trúc Chỉ mục:** Chỉ mục có thứ tự (B+-Trees) và chỉ mục băm; chỉ mục **chính/phân cụm** (quy định thứ tự vật lý; tối đa một trên mỗi tập tin) so với chỉ mục **phụ**.

– **Tối ưu hóa Truy vấn:** Dựa trên chi phí (thống kê) và các quy tắc kinh nghiệm (đẩy phép chọn/chiếu xuống sớm) để chọn kế hoạch hiệu quả.

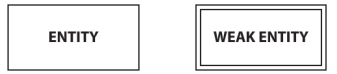
1.5 Các bước vẽ ERD

1. **Xác định Thực thể (Entities):**
   Tìm các *danh từ* (Nouns) quan trọng trong yêu cầu (Vd: Employee, Student). Tránh nhầm lẫn thuộc tính là thực thể.
2. **Xác định Mối kết hợp (Relationships):**
   Tìm các *động từ* (Verbs) kết nối các thực thể (Vd: Works_for, Teaches).
3. **Xác định Thuộc tính (Attributes):**
   Xác định thông tin chi tiết cho mỗi thực thể. Xác định thuộc tính đa trị, dẫn xuất, phức hợp.
4. **Xác định Khóa chính (Primary Keys):**
   Chọn thuộc tính định danh duy nhất cho mỗi thực thể và gạch chân nó.
5. **Xác định Bản số (Cardinality Ratio):**
   Phân tích số lượng tham gia: 1:1, 1:N, hay M:N.
6. **Xác định Ràng buộc tham gia (Participation):**
   Có bắt buộc không? (Total - Nét đôi) hay Tùy chọn? (Partial - Nét đơn).
   Hỏi: 'Thực thể A có thể tồn tại mà không cần B không?'
7. **Vẽ phác thảo & Tinh chỉnh:**
   Vẽ sơ đồ, loại bỏ các thuộc tính dư thừa. Chuyển quan hệ M:N thành thực thể liên kết nếu cần thiết.

1.6 Chen Notation: ER & EER

1.6.1 Entities & Attributes

Khi nào dùng: **Thực thể** cho đối tượng độc lập (NhanVien, SanPham, KhanhHang); **Thực thể yếu** cho đối tượng phụ thuộc (NguoiPhuThuoccủa nhân viên, ChiTietDonHang); **Khóa** là định danh duy nhất (MSNV, CCCD); **Đa trị** cho thuộc tính nhiều giá trị (số điện thoại, email); **Dẫn xuất** cho giá trị tính toán (tuổi từ ngày sinh, tổng tiền).



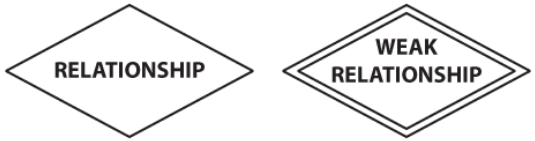
(Thực thể) (Thực thể yếu)

(Thuộc tính) (Khóa)

(Đa trị) (Dẫn xuất)

1.6.2 Relationships

Khi nào dùng: **Quan hệ thường** cho liên kết độc lập (NhanVien *làm việc cho* PhongBan, KhanhHang *mua* SanPham); **Quan hệ xác định** khi thực thể yếu phụ thuộc vào thực thể chủ (NguoiPhuThuoc *thuộc về* NhanVien với khóa bộ phận là tên người phụ thuộc).



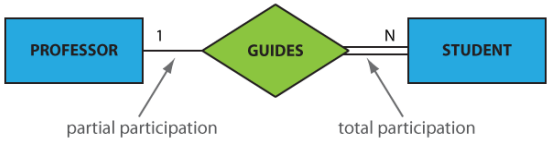
Bao gồm: Quan hệ (Hình thoi), Quan hệ xác định (Thoi đôi)

1.6.3 Ràng buộc (Constraints)

Khi nào dùng: Xác định quy tắc nghiệp vụ giữa các thực thể.

Bản số (Cardinality): 1:1 (NhanVien *quản lý* PhongBan - mỗi phòng có 1 trưởng), 1:N (PhongBan *có* NhanVien - nhiều nhân viên/phòng), M:N (NhanVien *tham gia* DUAN - nhiều-nhiều).

Tham gia (Participation):

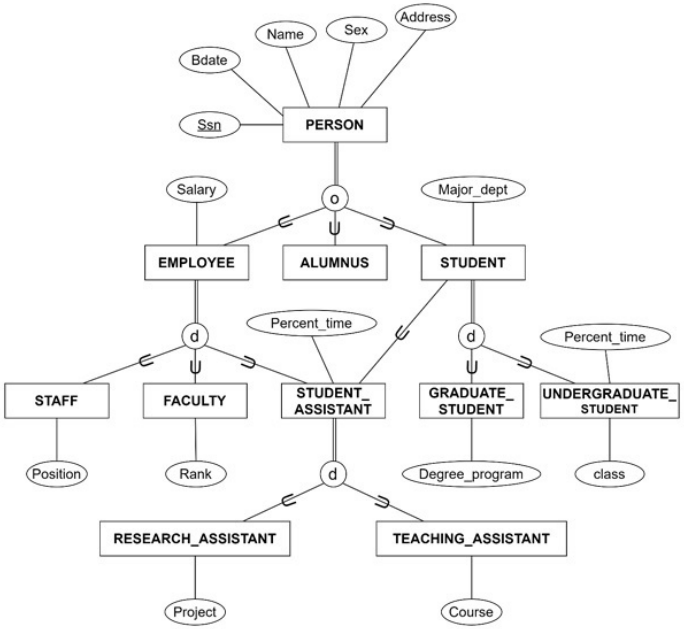


(Partial Participation (Từng phần): Nét đơn
Total Participation (Toàn phần): Nét đôi)

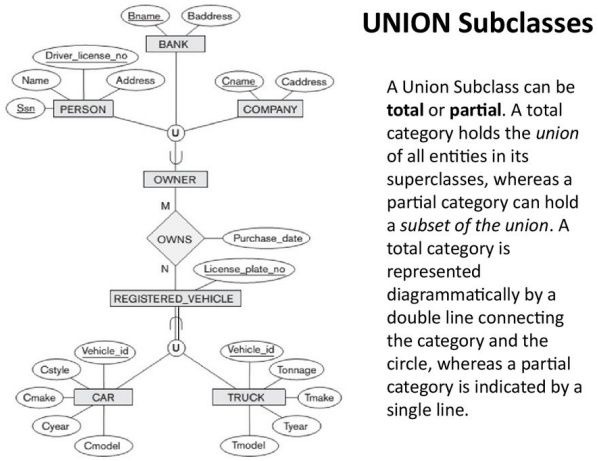
Min-Max (min, max): Ghi cặp số trên cạnh. Ví dụ: NhanVien $(1,1)$ *làm việc cho* $(0,N)$ PhongBan nghĩa là mỗi nhân viên bắt buộc làm việc cho đúng 1 phòng ban, mỗi phòng ban có thể có 0 đến nhiều nhân viên. $(0,1)$: tùy chọn, tối đa 1; $(1,N)$: bắt buộc, có thể nhiều. (An extension of Participation và Cardinality)

1.6.4 EER Chuyên biệt hóa & Tổng quát hóa

Khi nào dùng: **Disjoint** khi lớp con không chồng lấp (NhanVien là KỸ SƯ hoặc QUẢN LÝ, không đồng thời); **Overlapping** khi có thể thuộc nhiều lớp (NGƯỜI là SINH VIÊN và/hoặc NHÂN VIÊN); **Union** khi lớp con kế thừa từ nhiều lớp cha (CHỦ SỞ HỮU có thể là NGƯỜI hoặc CÔNG TY hoặc NGÂN HÀNG). **Total** khi mọi thực thể cha phải thuộc ít nhất 1 lớp con; **Partial** khi không bắt buộc.



(Disjoint & Overlapping)



(Union)

Ký hiệu: Hình tròn (d: disjoint, o: overlapping, U: union), Nét đôi (Total), Nét đơn (Partial).

2 Xử lý & Tối ưu hóa Truy vấn

*Query processing refers to the range of activities involved in extracting data from a database. Because SQL is declarative (specifying **what** to retrieve, not **how**), the DBMS must determine the most efficient execution strategy.*

2.1 Three Main Phases

1. **Parsing & Translation:** Translate high-level SQL query into internal relational-algebra representation. Verify syntax and semantics.
2. **Optimization:** Generate multiple equivalent execution plans. Estimate cost for each plan using statistics. Choose plan with minimum cost.
3. **Evaluation:** Execute chosen plan using query-execution engine. Apply evaluation primitives (algorithms for relational operations).

2.2 Phase 1: Parsing & Translation

2.2.1 Syntax & Verification

- **Syntax Checking:** Parser validates SQL syntax against language grammar.
- **Semantic Verification:** Verify relation names, attribute names, data types exist in database schema.
- **Authorization:** Check user has necessary privileges (SELECT, INSERT, UPDATE, DELETE) on referenced relations.

2.2.2 Internal Representation

- **Parse Tree:** Construct hierarchical syntax tree representing query structure.
- **Relational Algebra Translation:** Convert SQL to relational-algebra expression.
  Example: `SELECT name FROM instructor WHERE salary > 80000` $\rightarrow \pi_{name}(\sigma_{salary>80000}(instructor))$
- **Query Blocks:** Decompose complex SQL into basic units (`SELECT-FROM-WHERE` expressions) that map to algebraic operators.
  Nested Subqueries: Each subquery becomes separate query block. Correlated subqueries require special handling.
- **View Expansion:** If query references views, replace view names with their defining relational-algebra expressions.
  Recursive Process: Views may reference other views $\rightarrow$ expand until only base relations remain.

2.2.3 Query Tree (Initial Canonical Form)

- **Structure:** Tree representation where:
  - *Leaf nodes:* Base relations (tables).
  - *Internal nodes:* Relational operations (select σ, project π, join $\bowtie$, etc.).
  - *Execution flow:* Bottom-up evaluation - leaf data flows up through operations to root.
- **Canonical Form:** Initial tree follows SQL structure directly:

  - Cartesian products appear when multiple relations in FROM without explicit join conditions.
  - Selections appear high in tree (near root) reflecting WHERE clause position.
  - Projections at root reflecting SELECT clause.
- **Problem:** Canonical tree often inefficient → requires optimization transformation.

2.3 Phase 2: Tối ưu hóa Truy vấn (Query Optimization)

2.3.1 Tổng quan Quy trình Tối ưu hóa

After parsing, the optimizer transforms the canonical query tree into an efficient execution plan through two complementary approaches:
- **Heuristic Optimization:** Apply algebraic transformation rules to restructure tree (reduce search space).
- **Cost-Based Optimization:** Enumerate candidate plans, estimate costs using statistics, choose optimal plan.

2.3.2 Heuristic Optimization

Goal: Transform initial canonical query tree into more efficient equivalent tree using algebraic equivalence rules, without detailed cost estimation. This reduces search space for cost-based optimizer.

3 Core Heuristic Rules

1. **Perform Selections Early (σ down):** Push selection down close to leaf nodes (base relations).
   Reason: Reduce number of tuples before join → smaller intermediate results → less I/O.
   Benefit: If selection reduces 1M rows to 1K rows, subsequent join processes 1K instead of 1M.
2. **Perform Projections Early (π down):** Push projection down to reduce number of attributes (columns).
   Reason: Reduce "width"of intermediate relations → fewer bytes per tuple → save memory & I/O.
   Caution: Keep attributes needed by subsequent operations (join keys, selection predicates, final output).
3. **Avoid Cartesian Products ($\times$):** Combine Cartesian product + selection into Join ($\bowtie$).
   Reason: Cartesian product creates $|R| \times |S|$ tuples - extremely large intermediate relations.
   Example: 1000-row × 1000-row = 1M intermediate tuples. With join condition, result may be only 10K tuples.

Transformation Rules

- **Cascading of Selections:** $\sigma_{c1 \wedge c2}(R) = \sigma_{c1}(\sigma_{c2}(R))$.
  Application: Break conjunctive conditions to move parts down different branches.
- **Commutativity:** $\sigma_{c1}(R \bowtie S) = (\sigma_{c1}(R)) \bowtie S$ (if $c1$ only involves R).
  Application: Push selections down past joins.
- **Associativity:** $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$.
  Application: Reorder join sequence so most restrictive joins (highest selectivity) occur first.

Heuristic Algorithm (6 steps)

1. **Break up Selections:** Use cascading rule to split conjunctive select conditions.
2. **Move Selections Down:** Push σ down as far as possible (using commutativity).
3. **Reorder Leaf Nodes:** Rearrange leaf nodes (relations) - relation with most restrictive selection (lowest selectivity) first.
4. **Form Joins:** Combine Cartesian products with subsequent selections into Join operations.
5. **Move Projections Down:** Break projection lists, only keep attributes needed for query result or subsequent operations.
6. **Identify Subtrees:** Group operations that can be executed by single algorithm (e.g., single access method).

Trade-offs & Limitations

- **Not Always Optimal:** Heuristics reduce optimization cost but don't guarantee finding optimal plan.
  Example: "Selection early"usually good, but if selection needs to scan large relation while join has efficient index, joining first may be cheaper.
- **Hybrid Approaches:** Commercial optimizers combine both.
  System R: Heuristics (only consider left-deep trees) + cost-based (within those constraints).
  Oracle: Heuristics rank access paths + cost-based chooses join methods.

2.3.3 Cost-Based Optimization: Foundations

After heuristic optimization, system generates multiple candidate execution plans and estimates cost for each using statistical information from database catalog.

Cost Metrics

- **Primary Metric - Disk I/O:** Number of block transfers + number of disk seeks.
  Reason: For large disk-resident databases, I/O cost dominates CPU cost.
- **Secondary Metrics:** CPU time (for in-memory operations), network communication cost (distributed systems).
- **Response Time vs Throughput:** Systems may optimize for fast first-result (pipelining) or total execution time (materialization).

Cost Function Notation *Ước tính tài nguyên (disk I/O) để xử lý truy vấn: số block transfers (b) và số disk seeks (S).*
- b (hoặc b_r): Tổng số blocks trong file.
- r (hoặc n_r): Tổng số records (tuples) trong relation.
- s (hoặc s_A): **Selection cardinality** - số records trung bình thỏa điều kiện.
- bfr: Blocking factor (số records trên mỗi block).
- x (hoặc h_i): Số levels trong index (chiều cao B+-tree).
- t_S: Average time cho disk seek.
- t_T: Average time để transfer một block.
- **Công thức tổng:** Cost = $b \times t_T + S \times t_S$ (transfer time + seek time).

Statistical Information (Catalog Metadata)

- **Relation Statistics:** n_r (number of tuples), b_r (number of blocks), l_r (tuple size in bytes).
- **Attribute Statistics:** $V(A, r)$ (number of distinct values for attribute A), $min(A, r)$ and $max(A, r)$ (value range).
- **Index Statistics:** Index type, height (h_i), number of leaf blocks, clustering vs non-clustering.
- **Histograms:** Store distribution of attribute values to improve selectivity estimates.
  Equi-width: Divide value range into equal-sized buckets, count tuples in each.
  Equi-depth: Buckets contain approximately equal number of tuples.
  Use: Handle skewed data distributions (e.g., most employees earn 30K-50K, few earn >100K).

2.4 Cơ bản về Chỉ mục (Index Structures)

Indexes are critical data structures that enable efficient access paths for query execution. Understanding index characteristics is essential for cost estimation and access path selection.

2.4.1 Chỉ mục là gì?

- **Mục đích:** Tăng tốc độ truy xuất dữ liệu bằng cách tạo đường dẫn phụ trợ đến các bản ghi.
- **Khóa tìm kiếm:** (Các) thuộc tính dùng để tìm bản ghi; không nhất thiết là khóa chính; có thể là khóa phức hợp (nhiều cột).
- **Đánh đổi:** Đọc nhanh hơn so với ghi chậm hơn (chi phí bảo trì chỉ mục khi INSERT/UPDATE/DELETE).

2.4.2 Phân loại Chỉ mục

- **Theo Cấu trúc:**
  - *Có thứ tự (Ordered):* Các mục được sắp xếp (vd: B+-tree); hỗ trợ truy vấn khoảng.
  - *Băm (Hash):* Khóa được băm vào bucket; chỉ nhanh khi tìm kiếm chính xác (dấu bằng).
- **Theo Mật độ:**
  - *Đặc (Dense):* Một mục chỉ mục cho mỗi giá trị khóa tìm kiếm riêng biệt.
  - *Thưa (Sparse):* Một mục chỉ mục cho mỗi khối (hoặc mỗi giá trị phân cụm); nhỏ hơn, chi phí bảo trì thấp hơn.
- **Theo Thứ tự Vật lý:**
  - *Chỉ mục Chính/Phân cụm:* Khóa tìm kiếm quyết định thứ tự vật lý của tập tin (tối đa một per bảng); thường là chỉ mục thưa.
  - *Chỉ mục Phụ:* Đường dẫn truy cập thay thế độc lập với thứ tự vật lý; thường là chỉ mục đặc hoặc dùng gián tiếp cho các khóa không duy nhất.

2.4.3 B-Trees & B+-Trees

Khác biệt Cấu trúc *Mặc dù thường dùng thay thế cho nhau, B-Tree và B+-Tree có sự khác biệt quan trọng về nơi lưu data pointers.*

- **B-Tree:** Search key values appear *only once* in tree. Data pointers exist at *all levels* (internal + leaf nodes).
  → Lookup can terminate at internal node if key is found.
- **B+-Tree:** Data pointers only stored at *leaf nodes*. Internal nodes only contain separator keys to guide search.
  → Internal nodes contain more keys → higher fan-out → shorter tree height.
  → Leaf nodes linked together → efficient sequential access & range scans.
- **Industry Standard:** Most modern DBMS use B+-Tree (abbreviated as B-Tree) due to fewer disk I/Os.

General Structure & Properties

- **Balanced Tree:** Every path from root to leaf has same length (uniform height).
- **Node Capacity:** Each node = fixed-size disk block (typically 4KB or larger).
- **Fan-out:** Internal nodes contain from $\lceil n/2 \rceil$ to n children. High fan-out → B-Trees "fat and short".
  Example: 3-4 levels can store terabytes of data.
- **Leaf Nodes:** Contain from $\lceil (n-1)/2 \rceil$ to $n-1$ values, sorted by search key.
- **Separator Keys:** Internal node with keys $K_1, K_2, \ldots, K_{n-1}$: pointer P_i points to subtree with values X satisfying $K_{i-1} < X < K_i$.

Algorithms (Complexity: $O(\log_N M)$) *Search (Lookup):*

- **Traversal:** Start from root. At each internal node, binary search finds correct child pointer covering search key range.
- **Termination:** Repeat until leaf node. In B+-Tree, search leaf for specific key and associated record pointer.
- **Range Queries:** Leaf nodes linked (sibling pointers) → efficient range scan. Find starting key, traverse linked list.

Insertion:

1. **Locate Leaf:** Find leaf node where new key belongs.
2. **Space Available:** If leaf has space → insert entry in sorted order.
3. **Splitting (Overflow):** If leaf full:
   - Split into 2 nodes: first $\lceil n/2 \rceil$ entries in original, rest in new node.
   - Middle key promoted (copied) up to parent as separator.
4. **Propagation:** If parent also full → split propagates upward. Root split → new root → tree height increases by 1.

Deletion:

1. **Locate & Remove:** Find entry in leaf node and delete.
2. **Underflow:** If node $< \lceil n/2 \rceil$ full:
   - *Redistribute:* Borrow entries from sibling node (shift keys through parent).
   - *Merge (Coalesce):* If cannot redistribute → merge with sibling. Separator key in parent is deleted/demoted.
3. **Propagation:** Merge can propagate upward, potentially reducing tree height.

Capacity Calculation Example *Parameters:* Block size $B = 512$ bytes; key size $V = 9$ bytes; data pointer Pr $= 7$ bytes; tree pointer $P = 6$ bytes.

Step 1: Fan-out Calculation

- **B-Tree Internal Node:** Contains p tree pointers + $(p-1)$ keys + $(p-1)$ data pointers.
  $(p \times 6) + ((p-1) \times (7+9)) \leq 512 \Rightarrow 22p \leq 528 \Rightarrow p = 23$
- **B+-Tree Internal Node:** Contains p tree pointers + $(p-1)$ keys (no data pointers).
  $(p \times 6) + ((p-1) \times 9) \leq 512 \Rightarrow 15p \leq 521 \Rightarrow p = 34$
  Insight: B+-Tree fan-out 48% higher (34 vs 23) → shorter tree.
- **B+-Tree Leaf Node:** Contains p_{leaf} key/data pointer pairs + 1 next pointer.
  $(p_{leaf} \times (7+9)) + 6 \leq 512 \Rightarrow 16 \times p_{leaf} \leq 506 \Rightarrow p_{leaf} = 31$

Step 2: Estimate Total Capacity (69% Full)

- **B-Tree (3 levels):** Average 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 nodes $fo = 34 \times 0.69 \approx 23$; 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
- **Observation:** B+-Tree holds $\sim 4\times$ more entries at the same height due to lighter internal nodes.

Implementation & Optimizations

- **Write-Ahead Log (WAL):** B-Trees overwrite pages on disk. Crash during multi-page update (split) → corruption.
  → WAL writes modifications before updating tree pages → durability & atomicity.
- **Concurrency (Latches):** Multiple threads access index concurrently → use latches (lightweight locks).
  Latch Crabbing: Lock parent → lock child → release parent → high concurrency.
- **Bulk Loading:** Inserting records one-by-one → random I/O → inefficient.
  → Bulk load: Sort data → build tree bottom-up (sequential writes) → create parent nodes as needed.
- **Flash Storage (SSD):** Random reads fast, random writes expensive (erase-modify-write cycles).
  → LSM-Tree (Log-Structured Merge-Tree) preferred for write-heavy workloads, but B-Trees remain general-purpose standard.
- **Prefix Compression:** Internal nodes only store prefix of key (enough to distinguish subtrees) instead of full key → increases fan-out.
- **Right-Only Appends:** With auto-incrementing keys, allocate new node when rightmost leaf full instead of split → avoids half-empty pages.

2.4.4 Advanced Index Types

- **Composite Indexes:** Multi-column keys (e.g., (City, LastName)); support leftmost prefix queries (City), (City, LastName); order columns by selectivity.
- **Function-Based Indexes:** Indexes on expressions (e.g., `LOWER(email)`); queries must use the exact function to utilize the index.

2.4.5 Chỉ mục Băm & Bitmap

Chỉ mục Băm (Hash Indexes)

- **Sử dụng:** Tìm kiếm chính xác cực nhanh (truy vấn điểm); không hỗ trợ khoảng.
- **Xử lý đụng độ:** Dùng danh sách liên kết (chaining) với bucket tràn.
- **Biến thể động:** Băm mở rộng/tuyến tính (Extendible/Linear hashing) tăng trưởng dần mà không cần xây lại toàn bộ.

Chỉ mục Bitmap *Tối ưu cho thuộc tính có độ chọn lọc thấp (ít giá trị riêng biệt).*

- **Cấu trúc:** Đánh số thứ tự bản ghi (0, 1, 2, ...); mỗi giá trị riêng biệt có một bitmap; bit $i = 1$ nếu bản ghi i có giá trị đó.
- **Ví dụ (bảng 5 dòng):**
  - `gender='m'`: 10010 `gender='f'`: 01101
  - `income='L1'`: 11000 `income='L2'`: 00100
- **Truy vấn: gender='f' AND income='L2'**
  01101 AND 00100 = 00100 ⇒ bản ghi 2
- **Ưu điểm:** Gọn nhẹ (1 triệu dòng = 125 KB mỗi bitmap); thao tác bitwise nhanh; hiệu quả cho bộ lọc nhiều điều kiện; hỗ trợ COUNT qua đếm bit.

2.5 Hàm Chi phí cho Các Phép Toán (Operation Cost Functions)

2.5.1 Selection Cost Functions

Phép chọn (σ) có thể dùng quét file hoặc truy cập index. Chi phí chưa tính ghi kết quả.

S1: Linear Search (File Scan)

Quét mọi block và test tất cả records. Áp dụng cho mọi cấu trúc file.

- **Trường hợp chung: $\mathbf{C_{S1a} = b}$** blocks.
  Time: $t_S + b \times t_T$ (1 seek ban đầu + transfer all blocks).
- **Equality on Key: $\mathbf{C_{S1b} = b/2}$** blocks (trung bình, dừng khi tìm thấy).
  Time: $t_S + (b/2) \times t_T$.

S2: Binary Search

File đã sắp xếp theo search attribute. $\mathbf{C_{S2} = \lceil \log_2 b \rceil + \lceil s/bfr \rceil - 1}$ blocks.
Giải thích: $\log_2 b$ để locate first block + $\lceil s/bfr \rceil$ để retrieve matching blocks contiguous.

S3a: Primary Index (Equality on Unique Key)

File được sắp xếp vật lý theo search key. $\mathbf{C_{S3a} = x + 1}$ blocks.
Time: $(h_i + 1) \times (t_T + t_S)$ (traverse index tree + 1 data block access).
VD: B+-tree height 3 → 3 index blocks + 1 data block = 4 I/Os.

S3b: Hash Index (Equality)

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

S5: Clustering Index (Equality on Non-Unique Key)

File sắp xếp, matching records lưu contiguous. $\mathbf{C_{S5} = x + \lceil s/bfr \rceil}$ blocks.
Time: $h_i \times (t_T + t_S) + t_S + b_{matches} \times t_T$ (index + 1 seek to data + transfer).
VD: Tìm Dno=5 với 1000 matching records, bfr=10 → x + 100 blocks.

S6a: Secondary Index (Equality on Non-Unique Key)

File **không** sắp xếp theo attribute này → matching records phân tán.
$\mathbf{C_{S6a} = x + s + 1}$ blocks (worst case: mỗi record cần 1 random I/O).
Time: $(h_i + n) \times (t_T + t_S)$ với n = số records.
Nguy cơ: Nếu s lớn, secondary index đắt hơn linear scan do nhiều random seeks.

S6b: Secondary Index (Range Query)

$\mathbf{C_{S6b} = x + (b_{I1}/2) + (r/2)}$ (giả sử half file thỏa điều kiện).
Performance Implication: Nếu số records lớn, optimizer thích linear scan hơn secondary index.

Selections với Comparisons (Range Queries)

- **Clustering Index:** Hiệu quả - records contiguous.
  Cost: $x + b_{range}$ (find first + sequential scan).
- **Secondary Index:** Đắt - index leaves scan + random I/O cho mỗi record.
  Nếu range lớn: Prefer linear scan.

Complex Selections

- **Conjunctive (AND):** Choose single most restrictive access path (smallest selectivity) → check remaining conditions in RAM.
  Alternatively: Use multiple secondary indices → retrieve pointers → compute **intersection**.
- **Disjunctive (OR):** If 1 condition needs linear scan → entire query needs linear scan.
  If indices for all: Retrieve pointers → compute **union**.

Bảng Tóm tắt Chi phí SELECT

Algorithm	Mô tả	Block Accesses
S1	Linear Search	b
S2	Binary Search	$\lceil \log_2 b \rceil + \lceil s/bfr \rceil - 1$
S3a	Primary Index (Unique)	$x + 1$
S5	Clustering Index (Equality)	$x + \lceil s/bfr \rceil$
S6a	Secondary Index (Equality)	$x + s + 1$
S6b	Secondary Index (Range)	$x + (b_{I1}/2) + (r/2)$

Lưu ý: Chi phí thời gian = $b \times t_T + S \times t_S$ (tách biệt transfer và seek).

2.5.2 Ví dụ Tính toán Chi phí: Phép chọn trên EMPLOYEE

Kịch bản: Quan hệ EMPLOYEE có thông số:
- $r_E = 10{,}000$ records, $b_E = 2{,}000$ blocks, $bfr_E = 5$ records/block.
- **Available Access Paths:**
  - *Ssn* (Secondary, Unique Key): $x = 4$, $s = 1$.
  - *Dno* (Secondary, Non-Unique): $x = 2$, $s = 80$ (assumption: 125 depts, so 10,000/125 = 80).
  - *Salary* (Clustering, Non-Unique): $x = 3$, $s = 20$.
  - *Sex* (Secondary, Non-Unique): $x = 1$, $s = 5{,}000$ (assumption: 2 values, so 10,000/2 = 5,000).

Example 1: Equality on Unique Key

Query: $\sigma_{\text{Ssn}='123456789'}$(EMPLOYEE).
- **S1b (Linear Search):** $C = b_E/2 = 2{,}000/2 = 1{,}000$ blocks.
  Explanation: Scan half file on average until record found.
- **S6a (Secondary Index on Key):** $C = x + 1 = 4 + 1 = 5$ blocks.
  Explanation: Traverse 4-level index + retrieve 1 data block.
- **Decision:** Use S6a (5 blocks ≪ 1,000 blocks). *Speedup: 200x.*

Example 2: Equality on Non-Unique Key

Query: $\sigma_{\text{Dno}=5}$(EMPLOYEE).
- **S1a (Linear Search):** $C = b_E = 2{,}000$ blocks.
- **S6a (Secondary Index):** $C = x + s = 2 + 80 = 82$ blocks.
  Explanation: 2 index blocks + 80 random I/Os (records scattered across blocks).
- **S5 (If Clustering Index existed):** $C = x + \lceil s/bfr \rceil = 3 + \lceil 80/5 \rceil = 3 + 16 = 19$ blocks.
  Explanation: 3 index blocks + 16 contiguous data blocks (records stored together).
- **Decision:** Use S6a (82 blocks ≪ 2,000). *Note: Clustering index would be 4.3x better (19 vs 82).*

Example 3: Conjunctive Selection (Multiple Conditions)

Query: $\sigma_{\text{Dno}=5 \wedge \text{Salary}>30{,}000 \wedge \text{Sex}='F'}$(EMPLOYEE).
Strategy: Choose most selective access path to retrieve candidate set, then filter remaining conditions in memory.
- **Via Dno (S6a):** $C = x + s = 2 + 80 = 82$ blocks. *Retrieves 80 records.*
- **Via Salary Range (Clustering):** $C \approx x + (b_E/2) = 3 + 1{,}000 = 1{,}003$ blocks.
  Assumption: Half employees earn > 30,000, need to scan half file.
- **Via Sex (S6a):** $C = x + s = 1 + 5{,}000 = 5{,}001$ blocks. *5,000 random I/Os (very expensive).*
- **Linear Scan (S1a):** $C = 2{,}000$ blocks.
- **Decision:** Use Dno index (82 blocks) to retrieve 80 candidate records → filter Salary > 30,000 and Sex='F' in memory.
  Final result size estimated: $80 \times 0.5 \times 0.5 = 20$ records (assuming 50% pass each filter).

Example 4: Secondary Index vs Linear Scan Trade-off

Query: $\sigma_{\text{Sex}='F'}$(EMPLOYEE) (highly non-selective).
- **S6a (Secondary Index):** $C = x + s = 1 + 5{,}000 = 5{,}001$ blocks.
  Problem: 5,000 random seeks → extremely expensive. At $t_S = 10$ms, $t_T = 1$ms: $(1 + 5{,}000) \times 11 = 55{,}011$ms ≈ 55 seconds.
- **S1a (Linear Scan):** $C = 2{,}000$ blocks.
  Time: $1 \times 10 + 2{,}000 \times 1 = 2{,}010$ms ≈ 2 seconds.
- **Decision:** Use linear scan (2,000 blocks ≪ 5,001 blocks). *27x faster despite having index!*
  Key Insight: Secondary index only beneficial when selectivity is high ($s \ll b$).

2.5.3 Cost Functions for JOIN Operations

Notation

- b_r, b_s: Number of blocks (số khối) in relations r and s.
- n_r, n_s (or $|R|, |S|$): Number of tuples (số bộ) in relations r and s.
- js: Join selectivity (độ chọn lọc kết nối) - fraction of tuple pairs matching. For equi-join: $js \approx 1/\max(NDV(A), NDV(B))$.
- jc: Join cardinality (lực lượng kết nối) - number of result tuples = $js \times |R| \times |S|$.
- bfr_{result}: Blocking factor (hệ số khối) of result relation.
- n_B (or M): Number of available memory buffer blocks (số khối bộ đệm).
- h_i (or x): Height of index tree (chiều cao B+-tree).
- s_B: Selection cardinality - average matching records in index lookup.

J1: Nested-Loop Join Variants

- **Basic Nested-Loop:** For each record in outer relation r, scan entire inner relation s.
  Cost: $b_r + n_r \times b_s$ block transfers + $(n_r + b_r)$ seeks.
  Problem: Extremely expensive if n_r is large.
- **Block Nested-Loop:** Process block-by-block instead of row-by-row.
  Cost (Worst): $b_r + (b_r \times b_s)$ transfers + $2 \times b_r$ seeks.
  Cost (With n_B buffers): $b_r + \lceil b_r/(n_B - 2) \rceil \times b_s$ transfers.
  Strategy: Use $n_B - 2$ buffers for outer, 1 for inner, 1 for output. Use smaller relation as outer.

J2: Indexed Nested-Loop Join

Index exists on join attribute of inner relation s. Replace file scans with index lookups.

- **General Formula:** $b_r + (n_r \times$ cost of selection on $s)$.
  Explanation: Read outer r (b_r) + perform n_r index lookups on s.
- **Secondary Index:** $b_r + n_r \times (x + s_B + 1)$ blocks.
  s_B*:* Average matching records. Each record needs random I/O if non-unique.
- **Primary/Clustering Index:** $b_r + n_r \times (x + 1)$ blocks.
  Assumption: Unique keys or matching records stored contiguous.

J3: Sort-Merge Join

Both relations must be sorted on join attributes. Highly efficient if already pre-sorted.

- **If already sorted:** $b_r + b_s$ block transfers (read both once).
- **If not sorted:** Cost(Sort r) + Cost(Sort s) + $b_r + b_s$.
  Sort Cost: Typically $b \times (2\lceil \log_{n_B-1}(b/n_B) \rceil + 1)$ for external merge-sort.
- **Best Use Case:** Data already sorted, or when result needs to be sorted.

J4: Hash Join

Partition both relations by hash of join key. Matching tuples fall into same partition.

- **Standard Hash Join:** $3(b_r + b_s)$ block transfers.
  3 Passes: 1. Read/write partitions for r. 2. Read/write partitions for s. 3. Join partitions.
  Requirement: Partitions of smaller relation must fit in memory.
- **Hybrid Hash Join:** $b_r + b_s$ (best case).
  Optimization: If entire build relation fits in memory, skip disk writes for some partitions.

Writing the Result

Above costs only estimate processing. Must add write cost to get total cost.

- **Write Cost:** jc/bfr_{result} blocks.
- **Total Cost:** Processing Cost + $(js \times |R| \times |S|)/bfr_{result}$.

2.5.4 Example: EMPLOYEE ⋈ DEPARTMENT

Scenario: EMPLOYEE ($|E| = 10{,}000$, $b_E = 2{,}000$) $\bowtie_{Dno=Dnumber}$ DEPARTMENT ($|D| = 125$, $b_D = 13$).
Indices: Secondary on $E.Dno$ ($x = 2$, $s = 80$). Primary on $D.Dnumber$ ($x = 1$).
Parameters: $js = 1/125$, $jc = 10{,}000$, $bfr_{result} = 4$ → write cost = 2,500 blocks. $n_B = 3$ buffers.

- **J1: Block Nested-Loop (D outer):**
  $C = 13 + \lceil 13/1 \rceil \times 2{,}000 + 2{,}500 = 13 + 26{,}000 + 2{,}500 =$ **28,513** blocks.
  Explanation: Read D (13) + scan E 13 times (26,000) + write result (2,500).
- **J2a: Indexed Nested-Loop (D outer → E inner):**
  $C = 13 + 125 \times (2 + 80) + 2{,}500 = 13 + 10{,}250 + 2{,}500 =$ **12,763** blocks.
  Explanation: Read D (13) + 125 lookups on E (each: 2 index + 80 scattered records) + write.
- **J2b: Indexed Nested-Loop (E outer → D inner):**
  $C = 2{,}000 + 10{,}000 \times (1+1) + 2{,}500 = 2{,}000 + 20{,}000 + 2{,}500 =$ **24,500** blocks.
  Explanation: Read E (2,000) + 10,000 lookups on D (each: 1 index + 1 data block) + write.
- **J3: Sort-Merge (if already sorted):**
  $C = 2{,}000 + 13 + 2{,}500 =$ **4,513** blocks.
  Best case: Only read both relations once + write result.
  If not sorted: Add sorting cost (typically $\approx 2b \log b$ per relation).
- **J4: Hash Join:**
  $C = 3 \times (2{,}000 + 13) + 2{,}500 = 6{,}039 + 2{,}500 =$ **8,539** blocks.
  Explanation: 3 passes through both relations (partition + build + probe) + write result.

Decision Ranking

J3 (if sorted) < J4 < J2a < J2b < J1.

- **Best:** Sort-Merge if already sorted (4,513).
- **Good:** Hash Join if not sorted (8,539) - no sorting overhead.
- **Moderate:** Indexed Nested-Loop with small outer (12,763).
- **Avoid:** Block Nested-Loop (28,513) - only use when no index/memory.

2.5.5 Join Order & Left-Deep Trees

Optimization Decisions

- **Choose Join Order:** Most critical decision for multi-way joins.
  Problem: n relations have $n!$ possible join orders (exponential search space).
  Selectivity Estimation: $js \approx 1/\max(V(A,r), V(B,s))$ (assumption: uniform distribution, no correlation).
  Join Cardinality: $jc = js \times |R| \times |S|$ - estimate intermediate result size.
  Strategy - Dynamic Programming: Build optimal plans bottom-up:
  1. Find best access path for each base relation (1-way plans).
  2. Find best plan to join any 2 relations (2-way plans).
  3. Find best plan to join any 3 relations using optimal 2-way plans (3-way plans).
  4. Continue until n-way plan found.

  Cost Principle: If $(R \bowtie S)$ is part of optimal plan, then plan for $(R \bowtie S)$ must itself be optimal.

Left-Deep Trees

Common constraint in cost-based optimization to reduce search space while enabling efficient pipelined execution.

- **Structure:** In left-deep tree, right child of each join is always base relation (leaf), never intermediate result.
  Example: $((R \bowtie S) \bowtie T) \bowtie U$ - left-deep. $(R \bowtie S) \bowtie (T \bowtie U)$ - bushy (not left-deep).
- **Benefits:**
  - *Index Availability:* Right child is base table → can use indices on right child for indexed nested-loop join.
  - *Reduced Search Space:* Only $n!$ left-deep trees for n relations vs $O(4^n)$ for all bushy trees.
    System R optimizer: 4-table join searches 24 plans instead of 256.
  - *Pipelining-Friendly:* Left-deep trees naturally support pipelined evaluation (see Evaluation section).
    Tuples flow from one operation to next without materializing intermediate results to disk.
- **Trade-off:** May miss optimal bushy tree, but optimization time reduction + pipelining benefits usually outweigh this.
  Modern Systems: Some optimizers (PostgreSQL, Oracle) consider limited bushy trees for specific patterns.

2.6 Phase 3: Query Evaluation

2.6.1 Evaluation Primitives

Query-execution engine implements chosen execution plan using low-level evaluation primitives.

- **Annotated Operations:** Each relational operation in plan annotated with:
  - *Algorithm choice:* e.g., "use hash join", "use index nested-loop", "use external merge-sort".
  - *Access method:* e.g., "use B+-tree index on salary", "use file scan on instructor".
  - *Buffer allocation:* How many memory buffers allocated to this operation.
- **Iterator Model (Volcano/Pipeline):** Each operator implements:
  - `open()`: Initialize operator state, allocate resources.
  - `next()`: Return next tuple (on-demand production).
  - `close()`: Clean up, release resources.

  Composition: Parent operator calls `next()` on child operators to pull tuples up the tree.

2.6.2 Materialization vs Pipelining

Two fundamental strategies for evaluating expression trees.

Materialization (Eagerly Evaluate Subexpressions)

- **Strategy:** Evaluate operations one at a time, bottom-up. Store complete result of each operation as temporary relation on disk before executing next operation.
- **Process:**
  1. Evaluate lowest-level operations (selections on base relations).
  2. Write complete intermediate result to temporary file.
  3. Read temporary file as input to next operation.
  4. Repeat until root operation completes.
- **Advantages:**
  - Simple to implement - each operation independent.
  - Intermediate results reusable if needed multiple times.
  - Works for all operation types.
- **Disadvantages:**
  - *High I/O Cost:* Write + read temporary file for each intermediate result.
  - *Delayed Results:* Cannot produce output until all subexpressions evaluated.
  - *Disk Space:* Requires storage for potentially large temporary relations.
- **Example Cost:** Query with 5 operations → 4 temporary relations written + read = 8 extra disk operations.

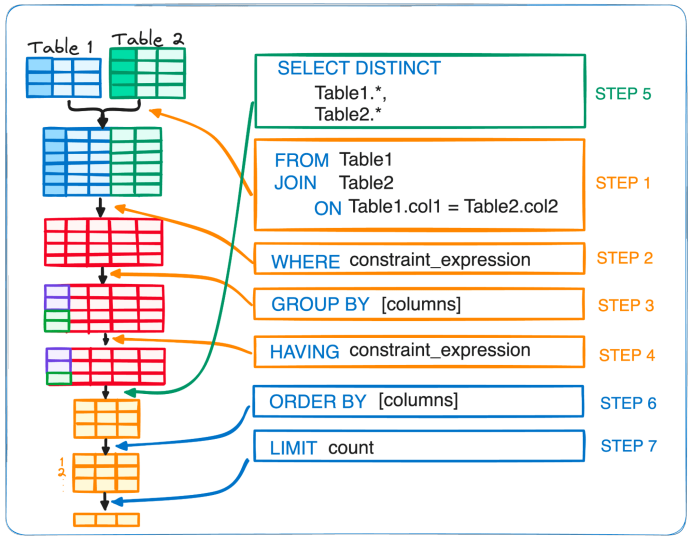
Pipelining (On-Demand Evaluation)

- **Strategy:** Pass tuples from one operation to next **without** storing intermediate results on disk. Operations combined into pipeline that runs concurrently.
- **Mechanism:** When parent operation needs tuple, it calls `next()` on child operation:
  - Child produces tuple → parent processes it → may produce output tuple.
  - Tuples flow through pipeline one-at-a-time or in small batches.
- **Demand-Driven (Pull Model):** Root pulls tuples from children, children pull from their children (iterator model).
  Alternative - Producer-Driven (Push): Each operation pushes tuples to parent as produced.
- **Advantages:**
  - *Eliminate I/O:* No temporary file writes/reads for intermediate results.
  - *Fast First Results:* System can start returning query results before complete evaluation finished.
  - *Memory Efficiency:* Only small buffer of tuples in memory, not entire intermediate relation.
- **Limitations (Pipeline Breakers):** Some operations require entire input before producing output:
  - *Sorting:* Must read all tuples to determine sort order.
  - *Aggregation (GROUP BY):* Must see all tuples in group to compute aggregate.
  - *Hash Join (Build Phase):* Must build complete hash table from inner relation.

  Solution: Use materialization for pipeline-breaking operations, pipelining elsewhere.
- **Left-Deep Trees & Pipelining:** Left-deep join trees enable long pipelines:
  Indexed nested-loop join: Outer tuples pipeline through join without materialization.
  Right child always base table → can use index lookups efficiently.

Hybrid Approach (Modern DBMS)

- **Strategy:** Combine both techniques - use pipelining where possible, materialize when necessary.
- **Example Query:** $\pi_{dept\_name,avg\_salary}(\sigma_{year=2017}(instructor) \bowtie department)$
  1. *Pipeline:* File scan on instructor → selection → join (if indexed nested-loop).
  2. *Materialize:* Join result temporarily stored (needed for aggregation).
  3. *Pipeline:* Aggregation → projection to final output.
- **Optimization Decision:** Optimizer estimates cost/benefit:
  - If intermediate result small → materialize (faster subsequent access).
  - If intermediate result large + used once → pipeline (save I/O).



(SQL Execution Order)

3 Big Data & Data Engineering

3.1 Đặc trưng Big Data (The Vs)

- **5V Cốt lõi (Core):**
  - **Volume:** Dung lượng khổng lồ (TB, PB, ZB).
  - **Velocity:** Tốc độ sinh ra & xử lý (Batch → Streaming).
  - **Variety:** Đa dạng định dạng (Structured, JSON, Video, Log).
  - **Veracity:** Độ tin cậy, tính xác thực (Messy/Noisy data).
  - **Value:** Giá trị chuyển hóa thành lợi ích kinh doanh.
- **Các V Mở rộng (Extended):**
  - **Variability:** Tính biến thiên (Ý nghĩa dữ liệu thay đổi theo ngữ cảnh/thời gian).
  - **Validity:** Tính hợp lệ (Dữ liệu có đúng định dạng/chuẩn để dùng không).
  - **Vulnerability:** Tính bảo mật (Dễ bị tấn công/rò rỉ).
  - **Volatility:** Độ bay hơi (Thời gian lưu trữ trước khi xóa/lưu trữ lâu dài).
  - **Visualization:** Khả năng trực quan hóa (Để con người hiểu được).

3.2 Paradigm: Batch vs Streaming

Hai mô hình xử lý dữ liệu cơ bản trong Big Data.

	Batch Processing	**Streaming Processing**
Đặc trưng	Xử lý dữ liệu tĩnh, lượng lớn theo lô.	Xử lý dữ liệu động, liên tục theo thời gian thực.
Độ trễ	Cao (minutes - hours).	Thấp (seconds - milliseconds).
Công cụ	Hadoop MapReduce, Apache Spark (Batch mode).	Apache Flink, Spark Streaming, Kafka Streams.
Use Case	ETL, báo cáo cuối ngày, ML training.	Real-time analytics, fraud detection, monitoring.
Ưu điểm	Xử lý hiệu quả khối lượng lớn, đơn giản.	Phản hồi nhanh, phát hiện sự kiện ngay lập tức.
Nhược điểm	Không real-time, lãng phí khi data nhỏ.	Phức tạp, khó debug, cần xử lý out-of-order.

Kiến trúc Lai (Hybrid):

- **Lambda Architecture:** Batch layer (chính xác) + Speed layer (real-time) + Serving layer. Phức tạp, duy trì 2 code base.
- **Kappa Architecture:** Chỉ dùng Streaming (đơn giản hóa). Mọi dữ liệu qua stream processor, replay từ Kafka khi cần.

3.3 Partitioning & Replication

3.3.1 Sao chép (Replication)

Mục đích: High Availability (HA) và giảm độ trễ đọc.

- **Single-Leader (Master-Slave):** Mọi ghi vào Leader, Leader chép sang Followers. *Dễ nhất quán, nhưng Leader là nút cổ chai.*
- **Multi-Leader:** Nhiều node chấp nhận ghi. *Tốt cho đa trung tâm dữ liệu, nhưng khó xử lý xung đột.*
- **Leaderless (Dynamo-style):** Ghi/Đọc gửi tới nhiều node. Dùng cơ chế **Quorum** để xác nhận:
  $w + r > n$ (Write nodes + Read nodes > Total replicas) → Đảm bảo đọc thấy dữ liệu mới nhất.

3.3.2 Phân mảnh (Partitioning/Sharding)

Mục đích: Scalability (Mở rộng dung lượng/băng thông).

- **Key Range Partitioning:** Chia theo khoảng khóa (A-C, D-F).
  → *Ưu:* Query theo khoảng (Range scan) hiệu quả.
  → *Nhược:* Dễ bị *Hotspot* (nếu user dồn vào vần A).
- **Hash Partitioning:** Băm khóa để chia đều ($hash(key)\%N$).
  → *Ưu:* Phân phối đều, tránh Hotspot.
  → *Nhược:* Mất khả năng Range Query (phải quét tất cả).

3.4 Định dạng Lưu trữ (File Formats)

Lựa chọn định dạng ảnh hưởng trực tiếp đến hiệu năng đọc/ghi.

3.4.1 Row-based vs. Column-based

- **Row-oriented (CSV, Avro):**
  - Lưu trữ tuần tự từng dòng.
  - *Ưu điểm:* Ghi nhanh (append), tốt khi truy xuất toàn bộ thông tin của 1 entity (OLTP).
  - *Nhược điểm:* Chậm khi tính toán tổng hợp (SUM, AVG) vì phải đọc cả dữ liệu không cần thiết.
- **Column-oriented (Parquet, ORC):**
  - Lưu trữ riêng biệt từng cột.
  - *Ưu điểm:* Nén cực tốt (do dữ liệu cùng kiểu), tối ưu cho OLAP (chỉ đọc cột cần thiết).
  - *Nhược điểm:* Ghi chậm, update tốn kém.

3.4.2 So sánh Avro, Parquet, ORC

Đặc điểm	Avro	Parquet
Mô hình	Row-based	Column-based
Schema	JSON (lưu trong file)	Binary (footer)
Tối ưu cho	Ghi nhiều (Write heavy)	Đọc nhiều (Read heavy)
Schema Evo	Rất tốt (Thêm/bớt field)	Hạn chế
Ecosystem	Kafka, Hadoop	Spark, Impala, Presto

3.5 Hadoop Ecosystem

Open-source framework for distributed storage and processing.

1. **HDFS (Storage):** Distributed file system.
   - *NameNode:* Manages metadata (block locations).
   - *DataNode:* Stores actual data blocks.
   - *Mechanism:* Splits files into blocks (128MB), replicates (x3) for fault tolerance.
2. **YARN (Resource Management):** "Operating system"of the cluster.
   - Distributes resources (RAM, CPU) to applications.
   - Allows multiple engines (Spark, MapReduce) to run on the same cluster.
3. **MapReduce (Processing):** Batch processing model - divide and conquer on a distributed cluster.
   - *Map:* Chia nhỏ & Gán nhãn. Input → Split → `<Key, Value>`.
   - *Shuffle:* Xáo trộn & Gom nhóm. Chuyển dữ liệu qua mạng, gom cùng Key.
   - *Reduce:* Tổng hợp. Xử lý danh sách Value của mỗi Key.

3.6 Công nghệ NoSQL (Storage Tech)

Các mô hình NoSQL cho use case khác nhau.

MongoDB (Document Store)

- **Mô hình:** Schema-on-read, lưu trữ JSON/BSON documents. Collections thay vì tables.
- **Ưu điểm:** Linh hoạt schema (mỗi doc có cấu trúc khác nhau), dễ scale horizontal (sharding), query mạnh (aggregation pipeline).
- **Architecture:** Replica Sets (HA), Sharding (scale-out), WiredTiger storage engine.
- **Use Case:** CMS, Mobile apps, Catalog, Real-time analytics. *VD: Forbes, eBay, Uber.*
- **Trade-offs:** Không ACID cross-document (trước v4.0), chiếm RAM nhiều.

Redis (Key-Value Store)

- **Mô hình:** In-memory key-value, data structures (String, Hash, List, Set, Sorted Set).
- **Ưu điểm:** Cực nhanh (< 1ms latency), atomic operations, support pub/sub, Lua scripting.
- **Persistence:** RDB (snapshot) hoặc AOF (append-only log). Có thể dùng cả 2.
- **Use Case:** Caching (session, query result), Message Queue (Celery), Leaderboard, Rate limiting. *VD: Twitter, GitHub, Stack Overflow.*
- **Trade-offs:** Giới hạn RAM, single-threaded (1 core), không có query phức tạp.

Cassandra (Wide-Column Store)

- **Mô hình:** Wide-column, mỗi row có thể có số cột khác nhau. Organize theo Column Family.
- **Ưu điểm:** Ghi cực nhanh (LSM Tree), linear scalability, masterless (P2P), multi-datacenter replication.
- **Architecture:** Consistent hashing (ring), tunable consistency (quorum), compaction strategies.
- **Use Case:** Time-series data, IoT sensor logs, Event logging, Messaging. *VD: Netflix, Apple, Instagram.*
- **Trade-offs:** Đọc chậm hơn (nhiều SSTable), không join, modeling phức tạp (query-first design).

Neo4j (Graph Database)

- **Mô hình:** Nodes (entities) + Relationships (edges) + Properties. Native graph storage.
- **Ưu điểm:** Traversal cực nhanh (follow pointers), query trực quan (Cypher), ACID transactions.
- **Architecture:** Index-free adjacency (mỗi node chứa pointer đến neighbors).
- **Use Case:** Social networks, Recommendation engines, Fraud detection, Knowledge graphs. *VD: LinkedIn, Walmart, eBay.*
- **Trade-offs:** Scale khó hơn NoSQL khác, không tốt cho bulk data processing.

3.7 Batch Processing

3.7.1 Apache Spark (Unified Analytics Engine)

Thay thế MapReduce với tốc độ cao hơn 100x (in-memory).

- **Core Concept:** RDD (Resilient Distributed Dataset) - immutable, partitioned, parallel.
  - *Transformations:* Lazy (map, filter, join) - tạo DAG.
  - *Actions:* Eager (collect, count, save) - trigger execution.
- **Components:**
  - *Spark SQL:* Query structured data (DataFrame/Dataset API).
  - *Spark Streaming:* Micro-batch streaming (DStream).
  - *MLlib:* Machine learning library (classification, clustering, etc.).
  - *GraphX:* Graph processing (PageRank, connected components).
- **Ưu điểm:** In-memory caching, lazy evaluation, DAG optimization, unified API (batch + streaming).
- **Nhược điểm:** Tốn RAM, không true streaming (micro-batch), overhead cho job nhỏ.
- **Use Case:** ETL, ML training, interactive analytics, log processing. *VD: Netflix, Uber, Airbnb.*

3.8 Streaming Processing

Xử lý dữ liệu liên tục, độ trễ thấp (Real-time).

3.8.1 Các chiến lược xử lý (Strategies)

- **Thời gian (Time Domain):**
  - *Event Time:* Thời gian sự kiện xảy ra (quan trọng nhất).
  - *Processing Time:* Thời gian hệ thống nhận được dữ liệu.
  - *Watermark:* Cơ chế xử lý độ trễ (data đến muộn) trong Event Time.
- **Cửa sổ (Windowing):**
  - *Tumbling:* Cố định, không chồng (vd: mỗi 5p).
  - *Hopping/Sliding:* Có chồng lấp (vd: 5p, trượt mỗi 1p).
  - *Session:* Dựa trên hoạt động người dùng (hết timeout thì đóng).
- **Đảm bảo (Guarantees):**
  - *At-most-once:* Gửi 1 lần, chấp nhận mất (vd: Log).
  - *At-least-once:* Không mất, chấp nhận trùng lặp.
  - *Exactly-once:* Chính xác 1 lần (Khó nhất, cần Flink/Kafka).

3.9 Giao thức IoT (IoT Protocols)

3.9.1 MQTT (Message Queue Telemetry Transport)

Design Philosophy

Được thiết kế cho **constrained devices (thiết bị hạn chế)** trong môi trường mạng khó khăn:

- **Low bandwidth (băng thông thấp):** Minimized data packets (gói dữ liệu tối thiểu).
- **High latency (độ trễ cao):** Tối ưu cho mạng chậm.
- **Unreliable networks (mạng không ổn định):** Xử lý mất kết nối, chập chờn.
- **Low power usage:** Tiết kiệm pin cho IoT devices.
- **Real-time processing:** Hỗ trợ xử lý real-time cho streaming data.

Architecture: Publish-Subscribe Model

Khác với client-server truyền thống (client connect trực tiếp client), MQTT dùng **pub-sub model**.

- **Broker:** Central component nhận published messages và deliver đến subscribers phù hợp.
  Flow: Client gửi `CONNECT` → Broker trả `CONNACK`.
- **Publisher:** Client publish message đến broker với specific topic.
- **Subscriber:** Client listen incoming messages trên particular topics.
- **Advantage:** Decoupling (tách rời) - publisher không cần biết subscriber, scale dễ dàng.

Topics (Routing Mechanism)

Structure: Hierarchical strings với topic levels phân cách bởi `/`.

Type	Pattern	Matches & Description
Exact	`area1/humidity`	Chỉ match chính xác topic này. *Use:* Specific sensor.
Single-level	`area1/+`	`+` matches *1* level bất kỳ. ✓ `area1/humidity`, `area1/light` × `area1/sensors/temp`
Multi-level	`area1/#`	`#` matches *tất cả* levels. ✓ `area1/humidity`, `area1/sensors/temp` ✓ `area1/zone2/light`
Root wildcard	`#`	Matches *everything* in broker. (Use with caution - high load)

Smart Farming Example: Publisher sends `area1/humidity:65%` → Subscribers với patterns `area1/humidity`, `area1/+`, hoặc `area1/#` đều nhận được.

Quality of Service (QoS) Levels

Để xử lý network reliability khác nhau, MQTT định nghĩa 3 QoS levels:

Level	Guarantee	Mechanism & Use Case
0	At most once	Delivered once, no confirmation. Fast, có thể mất message. *Use:* Sensor readings không critical.
1	At least once	Delivered ít nhất 1 lần, requires confirmation (ACK). Có thể duplicate. *Use:* Important data.
2	Exactly once	Four-step handshake. Chậm nhất nhưng không duplicate. *Use:* Financial transactions, critical commands.

Note: Actual QoS = min(Publisher QoS, Subscriber QoS).

Practical Application: Smart Farming

- **Sensors (Publishers):**
  - *DHT11:* Temperature & air humidity.
  - *BH1750:* Luminosity (độ sáng).
  - *DHT22:* Soil moisture (độ ẩm đất).
- **Flow:** Sensors publish data đến broker với topics như `area1/moisture` → Management system subscribes topics → Real-time monitoring & decision making.
- **Benefits:** Cost-effective, scalable, knowledge base creation cho farming management.

3.9.2 MQTT vs HTTP Comparison

Aspect	HTTP	MQTT
Model	Request-Response (Client-Server)	Publish-Subscribe (qua Broker)
Connection	Short-lived (đóng sau request)	Persistent (Keep-alive, lightweight)
Header Size	Large (KB range) Cồng kềnh metadata	Very small (2 bytes) Tiết kiệm bandwidth
Power	High consumption	Low power (tối ưu battery devices)
Network	Reliable networks	Unreliable networks (chập chờn, high latency)
Use Cases	• Web browsing • REST APIs • Large file transfers	• IoT sensors • M2M communication • Constrained devices

Key Insight: HTTP designed cho human-to-machine (web), MQTT designed cho machine-to-machine (IoT).

3.10 Pipelines & Orchestration

3.10.1 Apache Kafka vs Airflow

Apache Kafka	Apache Airflow
Loại: Event Streaming Platform (Message Broker).	**Loại:** Workflow Orchestration (Quản lý quy trình).
Đặc trưng: • Log bền vững (Durable log). • Decoupling (Tách rời). • Replayable, High throughput.	**Đặc trưng:** • Code-as-infra (Python DAGs). • Quản lý dependency phức tạp. • Backfill (chạy lại quá khứ).
Vai trò: "Xương sống"vận chuyển dữ liệu *Real-time*.	**Vai trò:** "Nhạc trưởng"điều phối Job (*Batch/ETL*).

3.10.2 CDC (Change Data Capture)

Theo dõi và đồng bộ thay đổi từ database nguồn sang đích.

- **Mục đích:** Real-time data replication, sync giữa OLTP và OLAP, event-driven architecture.
- **Cơ chế:**
  - *Log-based CDC:* Đọc database transaction log (binlog MySQL, WAL PostgreSQL). **Tốt nhất** - không ảnh hưởng source.
  - *Trigger-based:* Trigger trên INSERT/UPDATE/DELETE. Ảnh hưởng performance.
  - *Timestamp/Version-based:* Poll dựa trên `updated_at` column. Thiếu DELETE events.
- **Tools:** Debezium (Kafka Connect), AWS DMS, Oracle GoldenGate, Airbyte.
- **Use Case:** Sync OLTP → DW, Microservices data sharing, Cache invalidation, Audit logs.
- **Pattern:** Source DB → CDC Tool → Kafka → Sink (DW/Cache/Search).

3.10.3 ETL vs ELT

- **ETL (Extract-Transform-Load):** Transform *trước* khi vào kho. Schema-on-write. Dữ liệu sạch, bảo mật. (Truyền thống).
- **ELT (Extract-Load-Transform):** Load raw vào kho *trước*, transform sau. Schema-on-read. Tận dụng sức mạnh Cloud DW (BigQuery, Snowflake). (Hiện đại).

3.11 Kho Dữ liệu (Data Warehousing)

3.11.1 OLTP vs OLAP

	OLTP (Transactional)	OLAP (Analytical)
Mục tiêu	Vận hành hàng ngày (Operational).	Ra quyết định (Decision support).
Dữ liệu	Hiện hành, chi tiết, cập nhật liên tục.	Lịch sử, tổng hợp, đa chiều.
Truy vấn	Đơn giản, trả về ít dòng (Lookup).	Phức tạp, join nhiều, quét bảng lớn.
Thiết kế	Chuẩn hóa cao (3NF) để tránh dị thường.	Phi chuẩn hóa (Star/Snowflake) để đọc nhanh.
User	NV, App, Khách hàng.	Manager, Data Analyst.

3.11.2 4 Đặc trưng Chính (Inmon)

1. **Hướng chủ đề (Subject-oriented):** Tổ chức theo chủ đề chính (Khách hàng, Sản phẩm) thay vì theo ứng dụng (App Bán hàng, App Kho).
2. **Tích hợp (Integrated):** Dữ liệu từ nhiều nguồn được làm sạch, đồng nhất (đơn vị, format, encoding) trước khi nạp.
3. **Bất biến (Non-volatile):** Dữ liệu đã vào DW thì (thường) không bị sửa/xóa, chỉ đọc.
4. **Biến thiên theo thời gian (Time-variant):** Mọi dữ liệu đều gắn với mốc thời gian để phân tích xu hướng (Historical data).

3.11.3 Thách thức Xây dựng DW

- **Data Quality:** "Garbage In, Garbage Out". Dữ liệu nguồn bẩn làm sai lệch báo cáo.
- **ETL Complexity:** Tích hợp các hệ thống cũ (Legacy) rất phức tạp.
- **Performance:** Truy vấn phân tích tốn tài nguyên, cần tối ưu index/partition.
- **User Acceptance:** Người dùng không hiểu hoặc không tin tưởng dữ liệu.
- **Cost:** Chi phí lưu trữ và duy trì hạ tầng cao.

3.11.4 Mô hình hóa (Modeling)

- **Star Schema:** Fact ở giữa, Dimension xung quanh. Phi chuẩn hóa dimension. *Hiệu năng cao, dễ query.*
- **Snowflake Schema:** Chuẩn hóa dimension (tách nhỏ). *Tiết kiệm không gian, join phức tạp.*

3.11.5 DW vs DL vs Lakehouse

- **Data Warehouse:** Dữ liệu có cấu trúc, cho BI/Reporting.
- **Data Lake:** Dữ liệu thô (Raw), đa dạng, giá rẻ, cho ML/DS.
- **Lakehouse:** Kết hợp (Lưu trữ rẻ của Lake + Quản lý/ACID của Warehouse).

3.11.6 SCD (Slowly Changing Dimensions)

Tại sao cần? Để đảm bảo báo cáo lịch sử chính xác. Nếu KH chuyển từ HCM ra HN năm 2024, doanh số năm 2020 vẫn phải tính cho HCM.

Type	Chiến lược	Đặc điểm & Use Case
0	**Retain Original:** Giữ nguyên, không bao giờ sửa.	Dữ liệu gốc là chân lý. (VD: Ngày sinh).
1	**Overwrite:** Ghi đè giá trị mới lên cũ.	Không cần lịch sử. Sửa lỗi chính tả.
2	**Add Row:** Add new row + *Effective Date* + *Current_Flag*.	**Most standard**. Track complete history of changes.
3	**Add Column:** Thêm cột *Previous_Value*.	Chỉ cần biết giá trị liền trước. (Ít dùng).
4	**Add History Table:** Tách bảng lịch sử riêng (Mini-Dimension).	Tối ưu khi bảng chính quá lớn và chỉ một nhóm thuộc tính thay đổi nhanh.
5	**Hybrid (4 + 1):** Mini-dimension + "Current"reference in main table.	Optimize queries when need both detailed history and current value quickly.
6	**Hybrid (1 + 2 + 3):** Type 2 row + column containing current value (Type 1).	*"Pure Type 6"*: Helps query history while still easily group by current value.
7	**Hybrid (Dual Keys):** Fact table contains both *Surrogate Key* (history) and *Natural Key* (current).	Most flexible: Join by Surrogate to view history, join by Natural to view current.

Các kỹ thuật xử lý dữ liệu thay đổi theo thời gian (0 → 7).

3.12 Quản lý Dữ liệu

3.12.1 Vấn đề Tích hợp (Data Integration Issues)

- **Heterogeneous data sources:** Khác biệt về hệ quản trị cơ sở dữ liệu và định dạng tệp.
  VD: MySQL, PostgreSQL, Oracle, MongoDB, CSV, JSON, XML, Parquet.
- **Data Mapping:** Khác biệt về cấu trúc schema giữa các nguồn.
  VD: Bảng 'Employee' (Full_Name, DOB) vs 'Emp' (First_Name, Last_Name, Birth_Date).
- **Data Conflicts:** Conflicts in data type, value, format, unit, precision.
  Type: String vs Integer for employee code. *Format:* DD/MM/YYYY vs MM/DD/YYYY.
  Unit: USD vs VND, km vs miles. *Precision:* 2 vs 4 decimal places.
- **Data Redundancy:** Dữ liệu trùng lặp từ nhiều nguồn cần khử trùng (Deduplication).
  VD: Cùng khách hàng xuất hiện trong CRM và ERP với ID khác nhau.
- **Entity Resolution:** Xác định 2 bản ghi từ 2 nguồn khác nhau là cùng 1 thực thể.
  VD: 'Nguyen Van A' (DB1) và 'A Nguyen' (DB2), 'IBM' vs 'International Business Machines'.
  Kỹ thuật: Fuzzy matching, similarity scores (Levenshtein distance), Master Data Management (MDM).
- **Constraints Violation:** Constraint violations during data integration.
  Primary Key: Duplicate primary keys during merge. *Foreign Key:* References don't exist.
  Semantic: Invalid values (negative age, future dates).
- **Data Quality Issues:** Data quality problems during integration process.
  Accuracy: Deviates from reality. *Completeness:* Missing required fields (NULL).
  Uniqueness: Duplicates. *Timeliness:* Stale/outdated data. *Consistency:* Conflicts between sources.
- **Communication Heterogeneity:** Khác biệt về giao diện và giao thức truyền thông.
  VD: REST API vs SOAP, HTTP vs FTP, Batch files vs Real-time streams, GraphQL vs SQL.

3.12.2 Data Quality Dimensions

Các tiêu chí đánh giá chất lượng dữ liệu trong hệ thống thông tin.

6 Core Dimensions

1. **Accuracy (Chính xác):** Mức độ dữ liệu phản ánh đúng đối tượng/sự kiện thực tế.
   Định nghĩa: Sự không chính xác có nghĩa là hệ thống biểu diễn một trạng thái thế giới thực khác với trạng thái đáng lẽ phải được biểu diễn.
   VD: Địa chỉ sai, số điện thoại cũ, thông tin lỗi thời.
2. **Completeness (Đầy đủ):** Tỷ lệ dữ liệu được lưu trữ so với khả năng 100% hoàn chỉnh.
   Định nghĩa: Khả năng của hệ thống biểu diễn mọi trạng thái có nghĩa của hệ thống thế giới thực.
   VD: NULL values, missing fields, incomplete records (missing email, phone number).
3. **Consistency (Nhất quán):** Sự vắng mặt của sự khác biệt khi so sánh hai hoặc nhiều biểu diễn của cùng một thứ.
   Định nghĩa: Sự không nhất quán có nghĩa là ánh xạ biểu diễn là một-nhiều (one-to-many).
   VD: Tổng doanh thu trong bảng Orders ≠ tổng trong Report, tên KH khác nhau giữa CRM và ERP.
4. **Validity (Hợp lệ):** Dữ liệu hợp lệ nếu tuân thủ cú pháp của định nghĩa (format, type, range).
   VD: Email correct format (has @), age in range [0, 150], enum has allowed values.
5. **Timeliness (Kịp thời):** Mức độ dữ liệu phản ánh thực tế từ mốc thời gian yêu cầu.
   Định nghĩa: Độ trễ giữa thay đổi trạng thái thế giới thực và sửa đổi tương ứng trong hệ thống.
   VD: Giá cổ phiếu cập nhật delay 10s, báo cáo tồn kho cách 1 ngày (stale data).
6. **Uniqueness (Duy nhất):** Không có thứ nào được ghi lại nhiều hơn một lần dựa trên cách xác định thứ đó.
   VD: Không có duplicate records, một khách hàng chỉ có một ID duy nhất.

Extended Dimensions

- **Interpretability (Khả năng Diễn giải):** Liên quan đến tài liệu và metadata có sẵn để diễn giải chính xác ý nghĩa và thuộc tính của nguồn dữ liệu.
  VD: Data dictionary, schema documentation, column descriptions, business glossary.
- **Accessibility (Khả năng Truy cập):** Đo lường khả năng người dùng truy cập dữ liệu từ văn hóa, trạng thái/chức năng vật lý và công nghệ có sẵn của họ.
  VD: API with rate limit, role-based access (RBAC), multi-language support.
- **Usability (Khả dụng):** Đo lường hiệu quả, hiệu suất, sự hài lòng mà người dùng cụ thể cảm nhận và sử dụng dữ liệu.
  VD: Dashboard dễ hiểu, query response time nhanh, format phù hợp với use case.
- **Trustworthiness:** Đo lường mức độ đáng tin cậy của tổ chức trong việc cung cấp nguồn dữ liệu.
  VD: Clear data lineage, complete audit logs, data provenance.

3.12.3 Quản lý Thông tin

Quản lý dữ liệu như một tài sản chiến lược của doanh nghiệp.

6 Khía cạnh Quản lý Thông tin

1. **Information Collection (Thu thập):** Xác định và thu thập dữ liệu từ các nguồn khác nhau.
   Hoạt động: Data ingestion, ETL pipelines, API integration, web scraping, sensors/IoT.
   VD: Crawl web data, stream from Kafka, batch import from CSV, CDC from transactional DB.
2. **Information Organization (Tổ chức):** Cấu trúc hóa và phân loại dữ liệu để dễ quản lý và truy xuất.
   Hoạt động: Schema design, data modeling (Star/Snowflake), taxonomy, metadata management.
   VD: Design DW dimensions/facts, create data catalog, tag data by domain.
3. **Information Storage (Lưu trữ):** Chọn và triển khai hệ thống lưu trữ phù hợp với đặc điểm dữ liệu.
   Hoạt động: Choose DBMS (RDBMS, NoSQL, DW), partitioning, replication, backup strategy.
   VD: OLTP on PostgreSQL, OLAP on Snowflake, unstructured data on S3/Data Lake.
4. **Information Manipulation (Thao tác):** Thực hiện các thao tác CRUD và chuyển đổi dữ liệu.
   Hoạt động: INSERT/UPDATE/DELETE, data transformation, cleansing, enrichment, aggregation.
   VD: Chuẩn hóa địa chỉ, deduplicate records, merge datasets, derive calculated fields.
5. **Information Processing (Xử lý):** Phân tích và tính toán để tạo insights từ dữ liệu thô.
   Hoạt động: Query execution, analytics, ML training, reporting, dashboarding, data mining.
   VD: SQL queries, Spark jobs, BI reports, predictive models, real-time analytics.
6. **Information Protection (Bảo vệ):** Đảm bảo an toàn, riêng tư và tuân thủ quy định.
   Hoạt động: Access control (RBAC), encryption (at-rest/in-transit), audit logging, compliance (GDPR).
   VD: Mã hóa PII, masking sensitive data, role-based permissions, backup/disaster recovery.

Tổng thể: Information Management bao gồm Data Governance (quản trị), Data Quality, Master Data Management (MDM), Security, và Lifecycle Management. Mục tiêu cuối cùng là đảm bảo dữ liệu **tin cậy, an toàn, dễ truy cập** để tạo giá trị kinh doanh.

3.13 Distributed Systems Challenges

*Khác với hệ thống đơn, hệ phân tán đối mặt với **partial failures** - một số phần hỏng trong khi phần khác hoạt động.*

3.13.1 8 Fallacies (Ngụy biện) của Distributed Computing

Những giả định sai lầm mà lập trình viên thường mắc phải (L. Peter Deutsch, Sun Microsystems).

1. **The Network is Reliable:** Mạng không đáng tin cậy - switches hỏng, cables ngắt, packets bị mất/reorder.
2. **Latency is Zero:** Local call (ns/μs) ≠ Remote call (ms). Chain calls → latency tích lũy.
3. **Bandwidth is Infinite:** Băng thông hữu hạn. Stamp coupling (truyền dữ liệu thừa) gây bottleneck.
4. **The Network is Secure:** Attack surface tăng. Mọi endpoint phải được bảo mật.
5. **Topology Never Changes:** Topology thay đổi liên tục (upgrades, failures, scaling) → timeout/failures.
6. **Only One Administrator:** Nhiều admin quản lý các segment khác nhau (firewall, DB) → khó phối hợp.
7. **Transport Cost is Zero:** Serialization, marshalling, network infra đều tốn chi phí.
8. **Network is Homogeneous:** Mạng gồm nhiều vendor (routers, switches) → packet loss, anomalies.

3.13.2 Unreliability: Network & Time

- **Network Unreliability:**
  - *Timeouts:* No bounded time → must use timeout. Too short: false positive. Too long: delayed detection.
  - *Network Partition (Netsplit):* Group of nodes isolated → "Split brain"(2 leaders both accept conflicting writes).
- **Time Unreliability:** "Time is an illusion không có global clock.
  - *Clock Drift:* Quartz clock drifts due to temperature. Even NTP sync has discrepancies.
  - *Process Pauses:* GC or VM suspension → node pauses longer than lease timeout → declared dead but still thinks it's alive → data corruption.
  - *Ordering Issues:* Last Write Wins (LWW) using timestamp dangerous - clocks not synced → overwrites new value with old.

3.13.3 Consistency & Consensus Challenges

- **CAP Theorem:** Can only achieve 2/3: **C**onsistency, **A**vailability, **P**artition Tolerance. Partitions unavoidable → choose CP or AP.
  → NoSQL thường chọn AP (Sẵn sàng + Chịu lỗi) thay vì CP.
- **FLP Impossibility:** Async system cannot guarantee consensus termination if 1 node crashes → must use timeout.
- **Two Generals' Problem:** Không thể đạt common knowledge (chắc chắn đồng ý) qua kênh unreliable - ACK cuối cùng có thể bị mất.
- **BASE:** **B**asically **A**vailable, **S**oft state, **E**ventual consistency (Nhất quán cuối cùng - chấp nhận dữ liệu cũ tạm thời).

3.13.4 Failure Models

- **Crash Faults:** Process stops completely, never returns (simplest).
- **Omission Faults:** Process skips steps or doesn't send/receive messages (buffer overflow, congestion).
- **Byzantine Faults:** Process behaves arbitrarily/maliciously, sends wrong messages (hardest, blockchain).

3.13.5 Managing "Truth"in Distributed Systems

- **Truth by Majority:** Không node nào tin được view của chính mình. *Sự thật = Quorum quyết định.*
  VD: Node tưởng mình là leader, nhưng quorum khai tử (do GC pause) → node đó "đã chết".
- **Fencing Tokens:** Monotonic number for storage to reject writes from nodes holding expired locks (zombie nodes).

3.13.6 Thách thức Xử lý Dữ liệu Phân tán

- **Data Skew (Lệch dữ liệu):** One partition contains too much data.
  → *Consequence:* Straggler problem - job waits for slowest node.
  → *Solution:* Salting (add random prefix) to split hot key.
- **Shuffle:** Transfer data between nodes over network (Map → Reduce).
  → *Optimization:* Broadcast Join (copy small table to all nodes) to avoid shuffling large table.