Database Management Systems

Table of contents

1 Database Management Systems **16**
- 1.0.1 Welcome to DBMS (CS2001) Course . . . 16
- 1.0.2 Course Faculty . . . 16
- 1.0.3 Course Instructors: . . . 17
- 1.0.4 How to Use This Portal . . . 17

I Week-01 **18**

2 Introduction **19**
- 2.1 Key Terminologies . . . 19
- 2.2 Why DBMS? . . . 19

3 View of Data **21**
- 3.0.1 Data Abstraction . . . 21
- 3.0.2 Schema and Instance . . . 22
- 3.1 Database Languages . . . 22

4 Database Design **24**
- 4.1 Database Engine . . . 24
  - 4.1.1 Storage Manager . . . 24
  - 4.1.2 Query Processor . . . 25
  - 4.1.3 Transaction Management . . . 26

II Week-02 **27**

5 Relational Model & DDL **28**
- 5.1 What is a Relational Database? . . . 29
- 5.2 The Concept of Keys . . . 29
- 5.3 Relational Queries & The Pure Language . . . 30
- 5.4 Relational Operators: Slicing and Dicing . . . 30
  - 5.4.1 The Big Two: Select vs. Project . . . 30
  - 5.4.2 Other Important Operators . . . 31
- 5.5 What is SQL? . . . 32

5.6 Domain Types & Integrity Constraints . . . 32
5.6.1 Domain Types . . . 32
5.6.2 Integrity Constraints . . . 33
5.7 DDL: Building the Blueprint . . . 33
5.7.1 1. The `CREATE` Command . . . 33
5.7.2 2. The `ALTER` Command . . . 34
5.7.3 3. The `DROP` Command . . . 34

6 Data Manipulation (DML) & Basic Queries 35
6.1 DML: Furnishing the House . . . 36
6.1.1 1. `INSERT`: Adding New Records . . . 36
6.1.2 2. `UPDATE`: Modifying Existing Data . . . 37
6.1.3 3. `DELETE`: Removing Records . . . 37
6.2 The Anatomy of a Query . . . 38
6.2.1 The `SELECT` Clause in Detail . . . 38
6.2.2 The `ORDER BY` Clause . . . 38
6.3 The Danger of Cross Joins . . . 39
6.3.1 Fixing the Cross Join . . . 40

7 Advanced SQL Filtering, Sets & Aggregation 41
7.1 Advanced Filtering . . . 42
7.1.1 1. String Operations (`LIKE`) . . . 42
7.1.2 2. Advanced Predicates (`BETWEEN` and `IN`) . . . 42
7.2 Dealing with the Unknown: `NULL` . . . 43
7.2.1 The `IS` Operator . . . 43
7.2.2 Three-Valued Logic . . . 43
7.3 Set Operations . . . 44
7.4 Aggregation & Grouping . . . 45
7.4.1 `GROUP BY`: Sorting into Buckets . . . 45
7.4.2 `HAVING`: The Bouncer for Buckets . . . 45
7.5 The SQL Execution Order . . . 46

III Week-03 48

8 Nested Subqueries 49
8.1 Nested Subqueries . . . 49
8.2 Subqueries in the `WHERE` Clause . . . 49
8.2.1 Set Membership: `IN` / `NOT IN` . . . 49
8.2.2 Set Comparison: `SOME` and `ALL` . . . 49
8.2.3 Test for Empty Sets: `EXISTS` . . . 50
8.2.4 Test for Duplicates: `UNIQUE` . . . 51

8.3 Subqueries in the FROM Clause . . . 51
8.3.1 The `WITH` Clause . . . 52
8.4 Subqueries in the SELECT Clause . . . 52
8.5 Modifying the Database . . . 53
8.5.1 Deletion . . . 53
8.5.2 Insertion . . . 54
8.5.3 Updates . . . 54

9 Joins **56**
9.1 Sample Data . . . 57
9.2 Cross Join . . . 57
9.3 Inner Join . . . 58
9.4 Outer Join . . . 58
9.4.1 Left Outer Join . . . 58
9.4.2 Right Outer Join . . . 59
9.4.3 Full Outer Join . . . 59
9.4.4 Using Explicit Conditions . . . 59
9.5 Quick Reference: Join Types . . . 60

10 Intermediate SQL **61**
10.1 Views . . . 61
10.1.1 Why Views? . . . 61
10.1.2 Creating a View . . . 61
10.1.3 Views Built on Views . . . 62
10.1.4 View Expansion . . . 62
10.1.5 Updating Views . . . 62
10.1.6 Materialized Views . . . 63
10.2 Transactions . . . 63
10.3 Integrity Constraints . . . 64
10.3.1 On a Single Relation . . . 64
10.4 Referential Integrity . . . 64
10.4.1 Cascading Actions . . . 65
10.4.2 The Chicken-and-Egg Problem . . . 65
10.5 SQL Data Types and Schemas . . . 66
10.5.1 Built-in Date/Time Types . . . 66
10.5.2 Indexes . . . 66
10.5.3 User-Defined Types (UDT) . . . 67
10.5.4 Domains . . . 67
10.5.5 Large Object Types . . . 67
10.6 Authorization . . . 68
10.6.1 Forms of Authorization . . . 68
10.6.2 Granting Privileges . . . 68
10.6.3 Revoking Privileges . . . 68

10.6.4 Roles . . . 69
10.6.5 Authorization on Views . . . 69
10.6.6 Other Authorization Features . . . 70

11 Advanced SQL **71**
11.1 Why Go Beyond Plain SQL? . . . 71
11.2 SQL Functions . . . 71
11.2.1 Table Functions . . . 72
11.2.2 Procedures . . . 72
11.3 Procedural Language Constructs . . . 73
11.3.1 Loops . . . 73
11.3.2 Conditionals . . . 74
11.3.3 Exception Handling . . . 74
11.4 Triggers . . . 75
11.4.1 BEFORE vs. AFTER . . . 75
11.4.2 Row-Level vs. Statement-Level . . . 75
11.4.3 Example: BEFORE trigger (data cleanup) . . . 76
11.4.4 Example: AFTER trigger (cross-table maintenance) . . . 76
11.5 Using Triggers Wisely . . . 77
11.5.1 Good Uses . . . 77
11.5.2 Danger Signs . . . 77

IV Week-04 **78**

12 Relational Algebra and Calculus **79**

13 Introduction to Query Languages **80**

14 Relational Algebra (RA) **81**
14.1 Examples . . . 81
14.2 Division Operation . . . 82

15 Tuple Relational Calculus (TRC) **83**
15.1 TRC Examples . . . 83

16 Domain Relational Calculus (DRC) **84**
16.1 DRC Examples . . . 84

17 Entity-Relationship Model **86**

18 Introduction to Entity Sets **87**
18.1 Strong Entity Set . . . 87

18.2 Weak Entity Set . . . 87
18.2.1 Characteristics of Weak Entity Sets . . . 87

19 Attributes **89**
19.0.1 Types of Attributes . . . 89

20 Entity-Relationship (ER) Diagrams **91**

21 Mapping Cardinalities (Relationship Types) **92**
21.1 One-to-One (1:1) Relationship . . . 92
21.2 One-to-Many (1:N) Relationship . . . 92
21.3 Many-to-One (N:1) Relationship . . . 93
21.4 Many-to-Many (M:N) Relationship . . . 93

22 Total and Partial Participation **95**

23 Expressing Weak Entity Sets in ER Diagrams **96**

24 Advanced ER Concepts **97**

25 Ternary Relationships **98**

26 Aggregation **99**

27 Extended ER Features: Specialization and Generalization **100**
27.1 Disjointness Constraints . . . 100
27.2 Completeness Constraints . . . 100
27.2.1 1. Overlapping and Partial . . . 100
27.2.2 2. Disjoint and Partial . . . 101
27.2.3 3. Disjoint and Complete . . . 102

28 Summary **104**

V Week-05 **105**

29 Relational Design: Redundancy, Anomalies & Functional Dependencies **106**
29.1 Redundancy and Anomalies . . . 106
29.1.1 A Bad Schema . . . 106
29.1.2 Anomalies . . . 107
29.1.3 Why This Happens — and the Fix . . . 107
29.2 Functional Dependencies . . . 108
29.2.1 Definition . . . 108
29.2.2 Keys, Redefined Using FDs . . . 108
29.2.3 Trivial FDs . . . 108

30 Armstrong's Axioms & Closure of Attribute Sets **109**
30.1 Armstrong's Axioms . . . 109
30.1.1 Worked Example . . . 109
30.1.2 Derived Rules . . . 110
30.2 Closure of Attribute Sets . . . 110
30.2.1 How to Compute It . . . 110
30.2.2 Worked Example . . . 110
30.2.3 Superkeys and Candidate Keys, via Closure . . . 111
30.2.4 Prime and Non-Prime Attributes . . . 111
30.2.5 Formula: Maximum Number of Superkeys . . . 111
30.2.6 Case 1: Only 1 Candidate Key (Single Attribute) . . . 111
30.2.7 Case 2: Only 1 Candidate Key (Composite Key) . . . 112
30.2.8 Case 3: Every Single Attribute is a Candidate Key . . . 112
30.2.9 Case 4: Two Distinct Candidate Keys . . . 112
30.2.10 Generic Set Formula . . . 112
30.2.11 Mathematical Representation . . . 112
30.2.12 Why Attribute Closure Matters . . . 112

31 Extraneous Attributes, Equivalence & Canonical Cover **114**
31.1 Extraneous Attributes . . . 114
31.1.1 How to Test . . . 114
31.1.2 Example — Extraneous on the Left . . . 115
31.1.3 Example — Extraneous on the Right . . . 115
31.2 Equivalence & Canonical Cover . . . 115
31.2.1 Equivalence of FD Sets . . . 115
31.2.2 Canonical Cover . . . 115
31.2.3 How to Compute It . . . 116
31.2.4 Worked Example . . . 116

32 Lossless Join Decomposition & Dependency Preservation **117**
32.1 Lossless Join Decomposition . . . 117
32.1.1 Worked Example: Supplier-Parts . . . 118
32.2 Dependency Preservation . . . 119
32.2.1 A Quick Contrast . . . 119
32.2.2 How to Test It . . . 119
32.2.3 Worked Example . . . 120

VI Week-06 **121**

33 Introduction to Normalization **122**

34 What is Normalization? **123**

35 Why Normalize? The Data Anomalies **124**
35.0.1 1. Insertion Anomaly . . . 124
35.0.2 2. Deletion Anomaly . . . 124
35.0.3 3. Update Anomaly . . . 124

36 Key Prerequisites **125**
36.1 Functional Dependencies (FDs) . . . 125
36.2 Database Keys . . . 125
36.3 Attribute Types . . . 125

37 Normal Forms (1NF to 3NF) **126**

38 1NF (First Normal Form) **127**
38.0.1 **Example: Failing 1NF** . . . 127

39 2NF (Second Normal Form) **128**
39.0.1 **Understanding Partial Dependency** . . . 128
39.1 **2NF Solved Example** . . . 128

40 3NF (Third Normal Form) **130**
40.0.1 **Note:** . . . 130
40.1 **3NF Solved Example** . . . 130

41 BCNF and Multivalued Dependencies **131**

42 BCNF (Boyce-Codd Normal Form) **132**
42.0.1 **Note:** . . . 132
42.1 **BCNF Solved Example 1: Success** . . . 132
42.2 **BCNF Solved Example 2: Failure** . . . 133

43 Multivalued Dependency (MVD) **134**
43.1 **Visualizing MVD** . . . 134
43.2 **Trivial MVD** . . . 135

44 MVD Theory Rules **136**

45 4NF (Fourth Normal Form) **137**

VII Week-07 **138**

46 Application Programs and Architecture **139**
46.1 Application Programs . . . 139
46.2 The Three-Layer Architecture . . . 139
46.2.1 Presentation Layer (Frontend) . . . 140

46.2.2 Business Logic Layer (Middle Layer / Tier 2) . . . 141
46.2.3 Data Access Layer (Backend / Tier 3) . . . 141
46.3 Architecture Classification . . . 141
46.4 Web Applications and Scripting . . . 142
46.4.1 The World Wide Web . . . 142
46.4.2 URLs, URIs, and URNs . . . 142
46.4.3 HTML and HTTP . . . 142
46.4.4 Sessions and Cookies . . . 143
46.5 Scripting . . . 143
46.5.1 Client-Side Scripting . . . 143
46.5.2 **Server-Side Scripting** . . . 143
46.5.3 Servlets . . . 144
46.5.4 JSP (Java Server Pages) . . . 144
46.5.5 PHP . . . 145

47 SQL and Native Languages **146**
47.1 ODBC (Open Database Connectivity) . . . 146
47.2 JDBC (Java Database Connectivity) . . . 147
47.3 Connectionist Bridges . . . 147
47.4 Embedded SQL . . . 148
47.4.1 Key Concepts . . . 148
47.5 Connecting Python to PostgreSQL . . . 149
47.6 Key API Methods . . . 150
47.7 CRUD Operations in Python . . . 151

48 RAD, Application Security, and Mobile Apps **152**
48.1 Rapid Application Development (RAD) . . . 152
48.2 Application Performance . . . 152
48.2.1 Where Caching Happens . . . 152
48.3 Application Security . . . 153
48.3.1 SQL Injection . . . 153
48.3.2 Password Leakage . . . 154
48.3.3 Authentication . . . 154
48.3.4 Application-Level Authorization . . . 154
48.3.5 Audit Trails . . . 155
48.4 Challenges in Web Application Development . . . 155
48.5 Mobile Apps . . . 155
48.5.1 What is a Mobile App? . . . 155
48.5.2 Architecture of a Mobile App . . . 156
48.5.3 Types of Mobile Apps . . . 156

VIIIWeek-08 157

49 Algorithms & Data Structures 158
49.1 Algorithms and Complexity Analysis . . . 158
49.1.1 Basics and Definitions . . . 158
49.1.2 Motivation for Analysis . . . 158
49.1.3 Factors of Analysis . . . 158
49.1.4 Methodologies . . . 158
49.2 Data Structures . . . 159
49.2.1 Linear Data Structures . . . 159
49.3 Non-Linear Data Structures: Trees . . . 160
49.3.1 Binary Search Trees (BST) . . . 161
49.4 Search and Sorting . . . 161

50 Physical Storage Systems 162
50.1 Physical Storage Media . . . 162
50.1.1 The Storage Hierarchy Pyramid . . . 162
50.2 Inside a Magnetic Disk (HDD) . . . 164
50.2.1 Data Organization on the Disk . . . 165
50.3 Disk Performance Metrics . . . 165
50.3.1 1. Access Time . . . 165
50.3.2 2. Data Transfer Rate . . . 166
50.3.3 3. Reliability (MTTF) . . . 166
50.4 Emerging Technologies . . . 166

51 Disk Performance Math 167
51.1 Intuition Behind the Math . . . 167
51.1.1 1. Capacity of a Disk . . . 167
51.1.2 2. Time Period of One Rotation . . . 167
51.1.3 3. Transfer Rate . . . 167
51.1.4 4. Transfer Time . . . 167
51.1.5 5. Addressing Sectors . . . 168
51.2 Solved Examples . . . 168

52 File Organization & Buffering 171
52.1 File Structure and Organization . . . 171
52.1.1 The Data Hierarchy . . . 171
52.1.2 Record Management . . . 171
52.1.3 File Organization Types . . . 171
52.2 Storage Access and Buffer Management . . . 172
52.2.1 The Role of the Buffer Manager . . . 172
52.2.2 Replacement Policies . . . 172
52.2.3 Pinned Blocks . . . 173

IX Week-09 **174**

53 Indexing **175**
53.1 What is an Index? . . . 175
53.2 Ordered Indices: Primary vs. Secondary . . . 175
53.3 Dense vs. Sparse Indexes . . . 176
53.3.1 Dense Index . . . 176
53.3.2 Sparse Index . . . 176
53.4 Multilevel Index . . . 177
53.5 Index Updates . . . 178
53.5.1 On Deletion . . . 178
53.5.2 On Insertion . . . 179
53.6 Index Definition in SQL . . . 179
53.7 Guidelines for Indexing . . . 180
53.8 Formulas for Solving the Problem . . . 180

54 Tree-Based Index Structures **181**
54.1 Why Trees for Indexing? . . . 181
54.2 The 2-3-4 Tree: Motivation . . . 181
54.3 What Is a 2-3-4 Tree? . . . 181
54.4 Search in a 2-3-4 Tree . . . 182
54.5 Insertion in a 2-3-4 Tree . . . 182
54.5.1 Node Splitting . . . 183
54.5.2 Early vs. Late Splitting . . . 183
54.6 Insertion Example: Step by Step . . . 184
54.7 Deletion in a 2-3-4 Tree . . . 187
54.7.1 Deletion Example . . . 187
54.8 B+ Trees . . . 189
54.9 Structure . . . 189
54.10Search . . . 190
54.11Insertion . . . 191
54.12Deletion . . . 191
54.13B-Tree vs B+ Tree . . . 191

55 Hashing and Bitmap Indices **193**
55.1 What is Hashing? . . . 193
55.2 Static Hashing . . . 193
55.3 Handling Bucket Overflow . . . 194
55.4 Deficiencies of Static Hashing . . . 194
55.5 Dynamic Hashing (Extendable Hashing) . . . 194
55.5.1 Core Idea . . . 195
55.5.2 Lookup . . . 195
55.5.3 Insertion . . . 196

55.5.4 Deletion . . . 196
55.5.5 Extendable Hashing: Pros and Cons . . . 196
55.6 Bitmap Indices . . . 197
55.6.1 What They Are . . . 197
55.6.2 Answering Queries with Bitmaps . . . 197

X Week-10 **199**

56 Transactions and ACID Properties **200**
56.1 Introduction to Transactions . . . 200
56.1.1 The Big Challenges . . . 200
56.2 The ACID Properties . . . 200
56.2.1 1. Atomicity (All or Nothing) . . . 200
56.2.2 2. Consistency . . . 201
56.2.3 3. Isolation . . . 201
56.2.4 4. Durability . . . 201
56.3 A Classic Example: Bank Transfer . . . 201
56.4 Transaction States . . . 202

57 Schedules and Serializability **203**
57.1 What is a Schedule? . . . 203
57.1.1 Serial vs. Concurrent Schedules . . . 203
57.2 Serializability . . . 204
57.2.1 1. Conflict Serializability . . . 204
57.2.2 2. View Serializability . . . 206
57.3 Recovery and Cascading . . . 206
57.3.1 Recoverable Schedules . . . 206
57.3.2 Cascading Rollbacks . . . 207
57.3.3 Cascadeless Schedules . . . 207
57.4 TCL (Transaction Control Language) . . . 208

58 Concurrency and Locking **209**
58.1 The Need for Concurrency Control . . . 209
58.2 Lock-Based Protocols . . . 211
58.2.1 The Lock-Compatibility Matrix . . . 211
58.3 The Two-Phase Locking Protocol (2PL) . . . 211
58.3.1 Lock Conversions . . . 212
58.4 Stricter Variants of 2PL . . . 212
58.4.1 Strict Two-Phase Locking . . . 212
58.4.2 Rigorous Two-Phase Locking . . . 212

59 Deadlocks and Timestamps **214**
59.1 Understanding Deadlocks . . . 214
59.1.1 Starvation . . . 215
59.2 Deadlock Prevention . . . 215
59.2.1 1. The Wait-Die Scheme (Non-Preemptive) . . . 215
59.2.2 2. The Wound-Wait Scheme (Preemptive) . . . 215
59.3 Deadlock Detection . . . 216
59.4 Timestamp-Based Protocols . . . 216
59.4.1 Read and Write Rules . . . 216
59.4.2 Correctness and Drawbacks . . . 217

60 (Optional) Real-World Concurrency **218**
60.1 Theory vs. Practice . . . 218
60.2 Multi-Version Concurrency Control (MVCC) . . . 218
60.2.1 How does it work? . . . 218
60.3 Isolation Levels in Practice . . . 219
60.3.1 The Trade-off . . . 219

XI Week-11 **220**

61 Backup and Recovery **221**
61.1 What is Backup and Recovery? . . . 221
61.2 Types of Data to Back Up . . . 221
61.3 Backup Strategies . . . 221
61.3.1 Full Backup . . . 221
61.3.2 Incremental Backup . . . 222
61.3.3 Differential Backup . . . 222
61.3.4 Comparing the Three . . . 223
61.4 Hot Backup . . . 223

62 Log-Based Recovery **225**
62.1 How Data Moves: The Access Model . . . 225
62.2 Log-Based Recovery . . . 225
62.3 Database Modification Schemes . . . 226
62.4 Undo and Redo . . . 226
62.5 Deciding What to Do After a Crash . . . 227
62.6 Checkpoints . . . 227

63 RAID: Redundant Array of Independent Disks **229**
63.1 What is RAID? . . . 229
63.2 Three Core Techniques . . . 229
63.2.1 Mirroring . . . 229

63.2.2 Striping . . . 230
63.2.3 Parity . . . 230
63.3 Standard RAID Levels . . . 232
63.3.1 RAID 0 — Block Striping . . . 232
63.3.2 RAID 1 — Mirroring . . . 233
63.3.3 RAID 2 — Parity . . . 234
63.3.4 RAID 3 — Byte Striping + Parity . . . 235
63.3.5 RAID 4 — Block Striping + Parity . . . 236
63.3.6 RAID 5 — Distributed Parity . . . 237
63.3.7 RAID 6 — Dual Parity . . . 238
63.3.8 RAID 01 — Mirror of Stripes . . . 239
63.3.9 RAID 10 — Stripe of Mirrors . . . 240

XII Week-12 **242**

64 Query Processing and Cost **243**
64.1 Overview of Query Processing . . . 243
64.1.1 1. Parser and Translator . . . 243
64.1.2 2. Optimization . . . 244
64.1.3 3. Evaluation . . . 244
64.2 Measures of Query Cost . . . 244
64.2.1 The Cost Formula . . . 245

65 Join Algorithms **246**
65.1 Understanding Join Costs . . . 246
65.2 1. Nested Loop Join . . . 246
65.3 2. Block Nested Loop Join . . . 247
65.4 3. Indexed Nested-Loop Join . . . 248
65.5 Example Scenario . . . 248
65.5.1 Cost using Nested Loop Join . . . 248
65.5.2 Cost using Block Nested Loop Join . . . 249
65.5.3 Cost using Indexed Nested-Loop Join . . . 249

66 Query Optimization and Equivalence **250**
66.1 Query Optimization Example . . . 250
66.2 Equivalence Rules . . . 251
66.2.1 1. Selection Rules . . . 251
66.2.2 2. Projection Rules . . . 251
66.2.3 3. Join Rules . . . 252
66.2.4 4. Set Operation Rules . . . 252

67 Architecture, Scaling, and Big Data **253**
67.1 What do Database Applications Need? . . . 253
67.2 RDBMS Architecture . . . 253
67.2.1 1. Centralized Systems . . . 253
67.2.2 2. Client-Server Systems . . . 254
67.2.3 3. Parallel Systems . . . 254
67.2.4 4. Distributed Systems . . . 254
67.3 Scaling Databases: Horizontal vs. Vertical . . . 254
67.3.1 Vertical Scaling (Scaling Up) . . . 254
67.3.2 Horizontal Scaling (Scaling Out) . . . 255
67.4 The Rise of Big Data . . . 255
67.5 The CAP Theorem . . . 255
67.6 Enter NoSQL . . . 256
67.6.1 Types of NoSQL Databases . . . 256

XIIIPlayground **258**

68 SQL Playground **259**

69 Python-DB Playground **260**

1 Database Management Systems

1.0.1 Welcome to DBMS (CS2001) Course

This portal contains all the materials, interactive playgrounds, and textbook references for the Database Management Systems course.

1.0.1.1 Course Overview

- **Course ID:** BSCS2001
- **Course Type:** Programming
- **Course Credits:** 4 Credits
- **Pre-requisites:** None

1.0.1.2 Textbook

- **Title:** *Database System Concepts* (7th Ed.)
- **Authors:** Silberschatz, Korth, Sudarshan
- **Publisher:** McGraw-Hill

1.0.2 Course Faculty

Dr. Partha Pratim Das
Professor, Department of Computer Science and Engineering, IIT Kharagpur

1.0.3 Course Instructors:

Manidipa

Pravin

Bhaskar

Arshpreet

1.0.4 How to Use This Portal

The content is organized chronologically. Just expand the navigation menu on the left sidebar or click **Start Course** above to jump into specific topics.

Part I

Week-01

2 Introduction

2.1 Key Terminologies

- **Data:** This is just raw, unorganized information without any extra context.
- **Record:** A collection of related data fields packaged together to describe one single real-world object. For example, grouping an instructor's ID, their name, and their department into a single cohesive row.
- **File:** A collection of related records.
- **Database:** The collection of structured information relevant to an enterprise is referred to as the **database**.

A **Database Management System (DBMS)** is a collection of interrelated data along with a set of programs designed to store, manage and access the data efficiently and conveniently.

> A DBMS is typically used to manage data that is highly valuable, relatively large and accessed by multiple users at the same time. **Applications** : Sales, Manufacturing, Banking, Airlines, University etc.

2.2 Why DBMS?

Historically, companies kept their permanent records in separate flat text files managed by conventional operating systems. Doing this without a centralized DBMS creates massive problems:

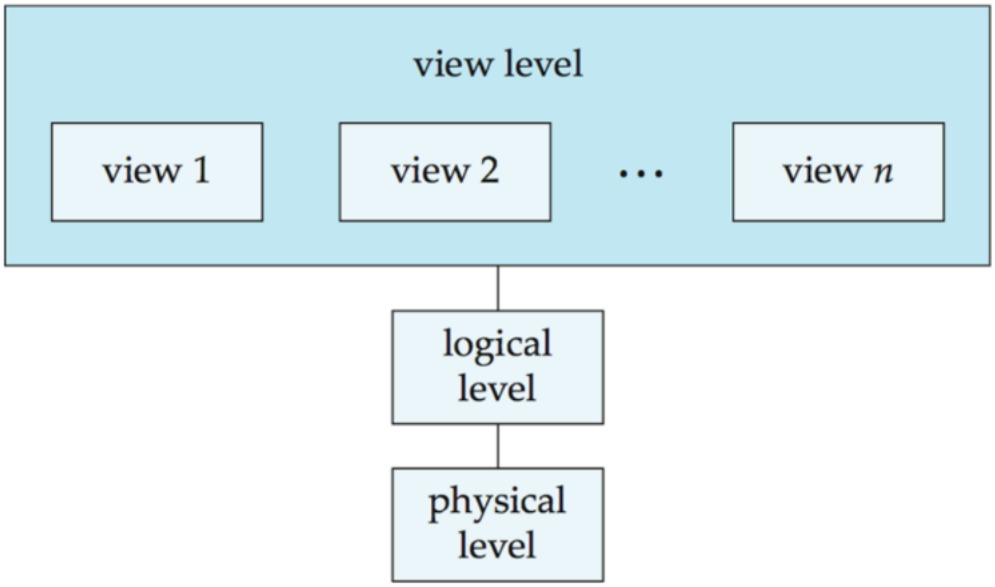
- **Data Redundancy & Inconsistency:** Separate departments build their own unique files over time. If a student changes their address, it gets updated in one file but remains outdated in another, leading to conflicting records.
- **Difficulty in Accessing Data:** Standard files aren't flexible. If a manager asks for a quick, unique list that the original developer didn't explicitly anticipate, a programmer has to write a brand-new script from scratch. It's slow and unresponsive.

- **Data Isolation:** Data gets scattered across disconnected files with conflicting file layouts and disparate formats, making it incredibly complex to build new applications that aggregate data.
- **Integrity Problems:** Real world rules, like a bank balance never dropping below zero, must be manually hardcoded into every single application script. If the rule changes, editing hundreds of decoupled programs is prone to human error.
- **Atomicity Issues:** Computer hardware is vulnerable to power failures or crashes. Database operations must be "all-or-nothing" (atomic). If a system crashes halfway through a fund transfer, a file system makes it very difficult to cleanly roll back the changes.
- **Concurrent-Access Anomalies:** To keep response times fast, systems must let multiple processes update data simultaneously. Without a centralized coordination engine, separate programs independently reading and writing raw text files will crash into each other and overwrite each other's updates.
- **Security Flaws:** Enforcing granular security (like letting a user see specific fields or rows but hiding others) is almost impossible because the underlying operating system doesn't understand the internal semantics of the data files.

3 View of Data

3.0.1 Data Abstraction

Abstraction refers to the hiding of implementation details to simplify the user interactions with the system.



- **Physical level** : Describes *how* the data is actually stored. It describes complex low-level data structures in detail.
- **Logical level** : Describes *what* data is stored in the database, along with the relationships that exist between them.
- **View level** : This is the highest level of abstraction that simplifies the user interaction by describing only the necessary part of the database. The system may provide different views for the same database.

3.0.2 Schema and Instance

- ***Schema*** refers to the overall design of the database. Database systems have several schemas, partitioned according to the levels of abstraction:
  - **Physical schema** : Describes the overall physical design of the database at the physical level.
  - **Logical schema** : Describes the overall logical design of the database at the logical level.

Name	Customer ID	Account #	Aadhaar ID	Mobile #

- ***Instance*** refers to the collection of information stored in the database at a given point of time.

Name	Customer ID	Account #	Aadhaar ID	Mobile #
Pavan Laha	6728	917322	182719289372	9830100291
Lata Kala	8912	827183	918291204829	7189203928
Nand Prabhu	6617	372912	127837291021	8892021892

- ***Physical Data Independence*** refers to the ability to modify the physical schema without changing the logical schema. This ensures that the different interfaces are defined such that any changes in one of the levels does not seriously influence others.
- ***Logical Data Independence*** refers to the ability to modify the logical schema without affecting the view level.

3.1 Database Languages

- **Data Definition Language (DDL)** :
  - The Data-Definition Language is used to specify the database schema and add structural properties or constraints to the data.
  - The processing of DDL statements generates metadata (data about data), which is stored securely in a centralized data dictionary. This directory is a special set of system tables that regular users cannot modify directly; the database system consults it before reading or altering any live data.
  - Example:

```
create table department(dept_name char(20),
                        building char(15),
                        budget numeric(12,2),
                        primary key (dept_name));
```

- **Data Manipulation Language (DML)** :
  - The Data-Manipulation Language enables users to access or alter data organized under the schema. DML handles information retrieval (queries), insertions, deletions, and updates.
  - Example:

```
select instructor.name
from instructor
where instructor.dept_name = 'History';
```

4 Database Design

Database design primarily involves designing the database schema to manage large, interconnected bodies of enterprise information.

- **User Requirements Specification** : Fully characterize and document the data needs of prospective database users.
- **Conceptual Design** : Translate user requirements into a high-level, detailed conceptual schema of the database, focusing entirely on data modeling rather than physical implementation constraints.
- **Logical Design** : Map the abstract, high-level conceptual schema onto the system-specific implementation data model of the chosen Database Management System (DBMS).
- **Physical Design** : Specify the internal and low-level physical features of the database configuration.

4.1 Database Engine

A database system is partitioned into independent modules that isolate core responsibilities. The database engine can be broadly divided into three main functional segments: the storage manager, the query processor, and the transaction manager.

4.1.1 Storage Manager

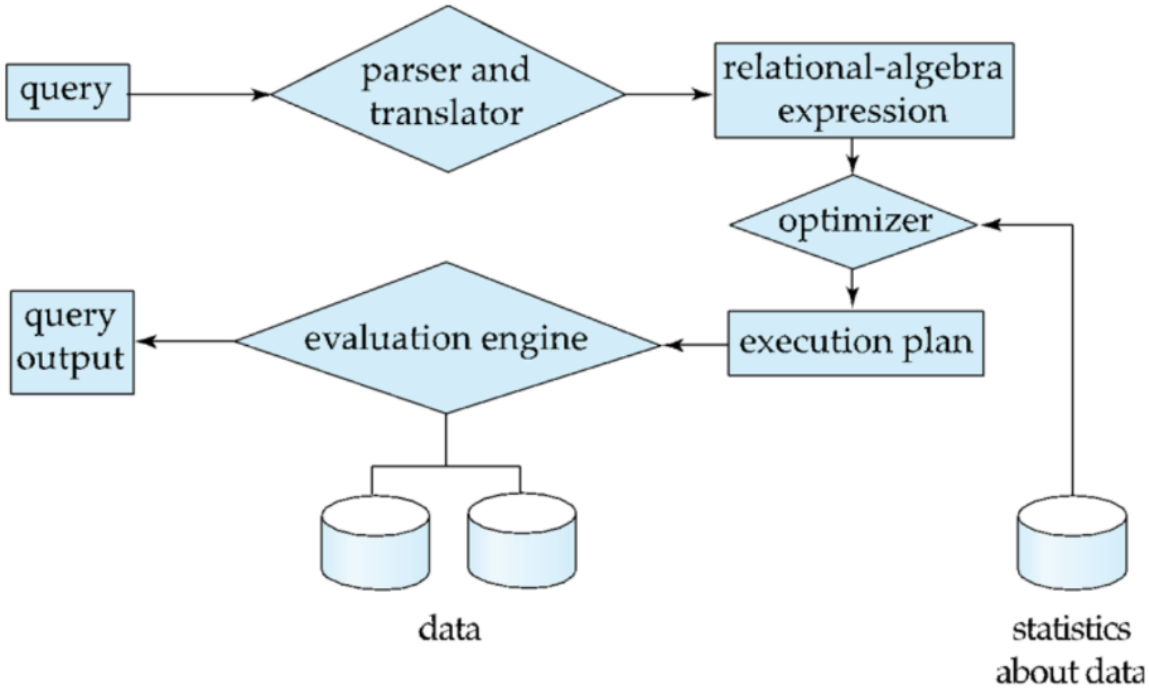
The storage manager acts as the operating interface between the low-level data stored on disk hardware and the application programs or queries submitted to the system.

- Validates user access permissions and tests whether newly modified data satisfies established consistency constraints.
- Keeps the global database in a correct state despite hardware crashes and prevents operational conflicts between concurrent tasks.
- Directs the physical allocation of disk space and manages the underlying byte structures used to represent database records.

- Coordinates caching strategies, determining when to fetch data from physical storage into volatile main memory so data pools can scale larger than the computer's actual RAM capacity.

4.1.2 Query Processor

The query processor abstracts execution details so developers can focus on writing high-level declarative commands, translating text into optimized sequences of lower-level physical operations.



- **Parsing and Translation** : When you submit a query, the system can't read it as raw text. The DDL Interpreter handles structural commands and updates the data dictionary. For data modifications, the DML Compiler parses your syntax for correctness and translates the declarative text into a relational-algebra expression (an internal mathematical representation the system can work with).
- **Optimization** : A single query can be executed using many alternative strategies, called evaluation plans. They all give the exact same output, but some consume drastically less hardware resources. The Query Optimizer calculates the estimated computational cost (disk I/O and CPU time) for each plan and programmatically selects the absolute most efficient one.

- **Evaluation** : The execution layer takes over. The Query Evaluation Engine takes those optimized, low-level instructions, executes them directly against the physical database files, and returns a clean result table back to the user.

4.1.3 Transaction Management

A transaction is a collection of separate program operations that perform a single logical function within an application. The transaction manager allows developers to treat multiple database updates as a single unit of work by enforcing the core ACID properties.

- Maintains system safety and manages failure recovery. It automatically detects system crashes and uses logging protocols to reverse or restore data back to the clean state that existed prior to the failed transaction.

Part II

Week-02

5 Relational Model & DDL

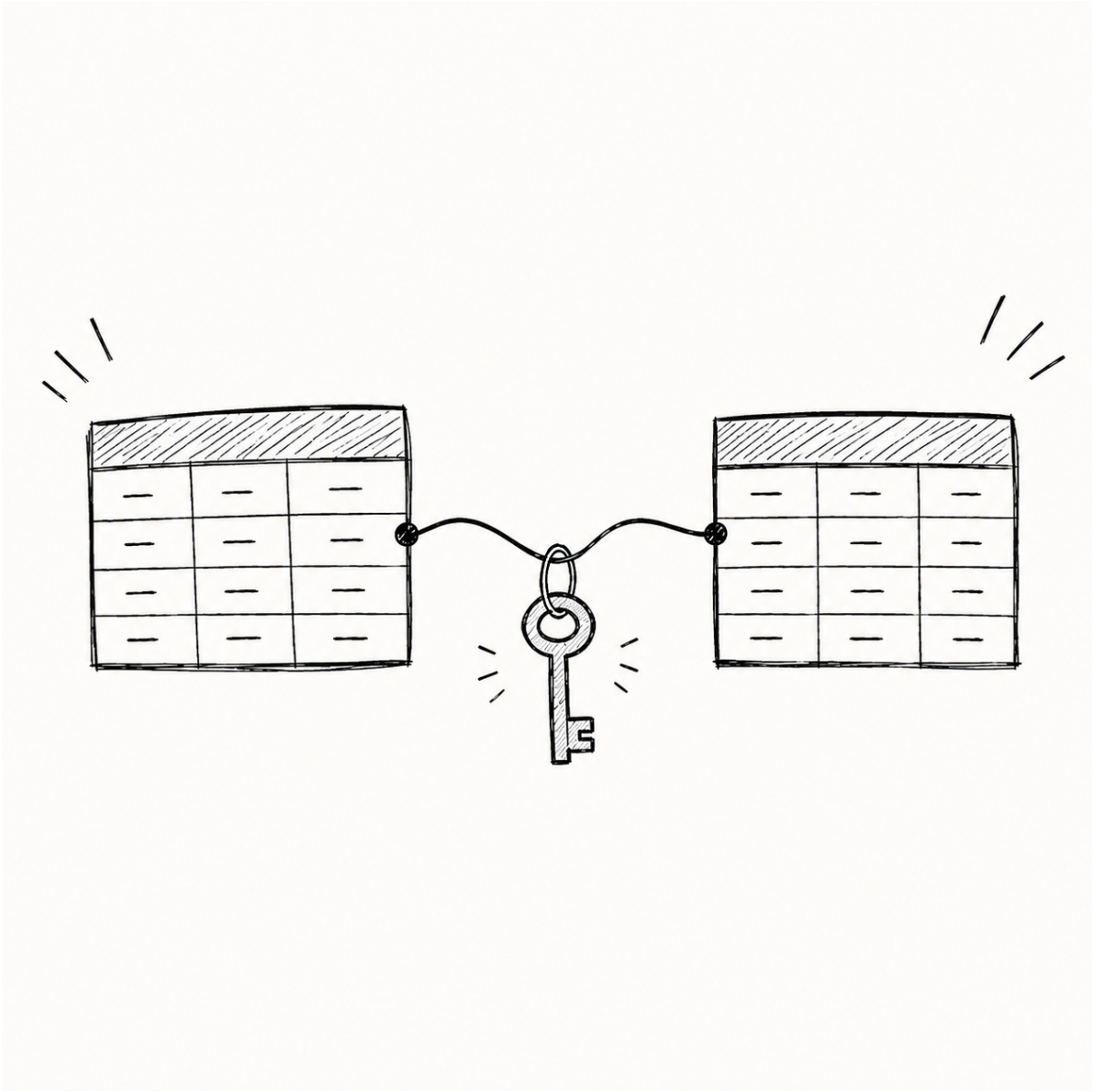


Figure 5.1: Relational Database Concepts

5.1 What is a Relational Database?

Before diving into commands, let's build some intuition. Imagine a school register. You have a list of students, their roll numbers, and the courses they are taking. In a **Relational Database**, we store data in structured **tables** (also called *relations*). Every row represents a single entity (like one specific student), and every column represents an attribute of that entity (like their name or grade).

The true power of relational databases isn't just storing tables—it's how we can *relate* them using Keys, and slice them using mathematical operations!

5.2 The Concept of Keys

In a database, we need a way to uniquely identify every row so we don't confuse two students named "John Smith". We do this using **Keys**.

- **Super Key**: Any combination of columns that can uniquely identify a row. For a student, `[Roll Number, Name]` is a super key, but it's bulkier than it needs to be.
- **Candidate Key**: A *minimal* Super Key. It has no unnecessary columns. Both `[Roll Number]` and `[Email Address]` could be candidate keys.
- **Primary Key**: The one Candidate Key you officially choose to identify your rows. Once chosen, no two rows can have the same Primary Key, and it cannot be empty.
- **Surrogate Key**: Sometimes, natural data (like names) makes for bad keys. A Surrogate Key is an artificially generated ID (like an auto-incrementing `Student_ID = 1, 2, 3...`) created purely to act as the Primary Key.
- **Foreign Key**: This is the literal glue that makes a database "relational"! A Foreign Key is a column in Table A that points exactly to the Primary Key in Table B. It links the two tables together.

Example: The `student_id` column in the `Grades` table is a Foreign Key that points to the `student_id` Primary Key in the `Student` table.

5.3 Relational Queries & The Pure Language

How do we ask a database for information? We use a **Query Language**.

There's a fundamental difference in how these languages work:

- **Imperative Languages** (like Python or Java): You must write step-by-step instructions on *how* to find the data.
- **Declarative Languages**: You simply describe *what* data you want, and the system figures out the best way to get it.

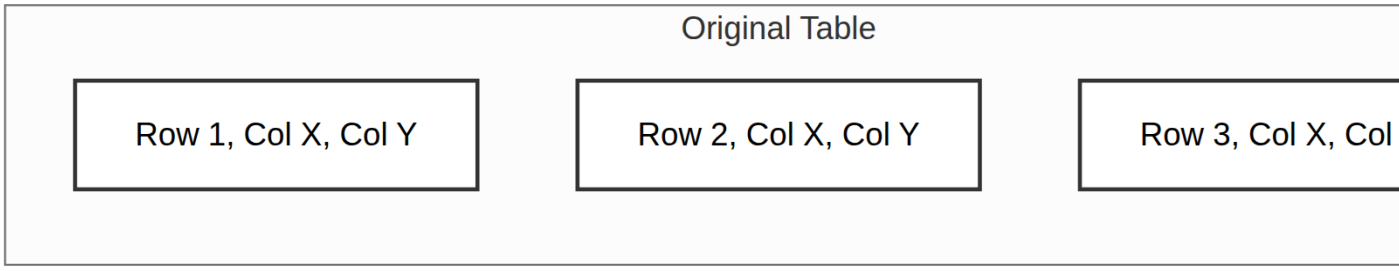
Before modern SQL existed, computer scientists created "pure" theoretical languages to prove databases could work. **Relational Algebra** is one of these pure languages. It uses mathematical symbols to describe exactly how to slice and dice tables.

5.4 Relational Operators: Slicing and Dicing

Relational Algebra gives us **Relational Operators**—surgical tools to extract what we need.

5.4.1 The Big Two: Select vs. Project

- **Select (σ)**: Filters **Rows**. It scans horizontally. (e.g., "Find all students who got an 'A' ").
- **Project (π)**: Filters **Columns**. It scans vertically. (e.g., "I only need to see Names and Roll Numbers").



5.4.2 Other Important Operators

Operator	Symbol	Intuition	Everyday Example
Union	$\cup$	Combines two sets into one big list.	"Give me the list of students in the morning class PLUS the evening class."
Intersection	$\cap$	Returns items present in *both* sets.	"Who is taking BOTH Physics AND Chemistry?"
Set Difference	$-$	Returns items in the first set but NOT in the second.	"Students taking Physics but NOT taking Chemistry."
Cartesian Product	$\times$	Combines *every* row of one table with *every* row of another.	Matching every color of a shirt with every size available.

Operator	Symbol	Intuition	Everyday Example
Natural Join	$\bowtie$	Connects two tables seamlessly based on matching columns.	Linking the `Student` table to the `Grades` table.

5.5 What is SQL?

While Relational Algebra is great for theory, we needed a real, practical language to type into a computer. Enter **SQL** (Structured Query Language).

A Brief History: Developed at IBM in the 1970s by Donald Chamberlin and Raymond Boyce, SQL was initially called "SEQUEL". It was designed to manipulate data stored in IBM's quasi-relational database management system. Over the decades, it became the undisputed, standardized language for relational databases globally.

SQL is broadly divided into three main categories:

1. **DDL (Data Definition Language)**: Used to define the structure (the blueprint).
2. **DML (Data Manipulation Language)**: Used to insert, update, and query the actual data (the furniture).
3. **DCL (Data Control Language)**: Used to manage permissions and security (the bouncers).

5.6 Domain Types & Integrity Constraints

Before building our table, we must set the rules.

5.6.1 Domain Types

We must tell the system *what kind* of data a column stores. This prevents someone from typing "Twenty" into an Age column.

Type	Description	Example
`char(n)`	Fixed-length text. Good for Zip Codes.	`char(5)` pads "Hi" to `"Hi "`
`varchar(n)`	Variable-length text. Good for names.	`varchar(50)` stores "Hi" as `"Hi"`
`int`	A whole number.	`25`
`numeric(p,d)`	A decimal number. `p` is total digits, `d` is after the decimal.	`numeric(4,2)` allows `44.22` but rejects `100.22`

5.6.2 Integrity Constraints

Constraints are extra rules we apply to columns to ensure the data makes sense.

- `NOT NULL`: The column cannot be left empty.
- `UNIQUE`: No two rows can have the same value in this column.
- `CHECK(condition)`: Ensures the value meets a specific requirement (e.g., `CHECK (age >= 18)`).
- `PRIMARY KEY`: Automatically enforces both `UNIQUE` and `NOT NULL`.

5.7 DDL: Building the Blueprint

Intuition: DDL is the architect. You use DDL to draw the blueprints and set the rules. DDL does **not** handle the actual data.

5.7.1 1. The `CREATE` Command

We use `CREATE TABLE` to bring a new table into existence. Let's create a `student` table with constraints!

```
CREATE TABLE student (
    roll_no varchar(10),
    name varchar(50) NOT NULL,
    age int CHECK (age > 10),
    email varchar(100) UNIQUE,
    primary key (roll_no)
);
```

What happened here?

- `name` cannot be left blank (`NOT NULL`).
- `age` must be greater than 10. If someone tries to insert a 9-year-old, the database will block it!
- `email` must be `UNIQUE` across all students.
- `roll_no` is our `PRIMARY KEY`.

5.7.2 2. The `ALTER` Command

What if we built our table, but later realized we forgot to add a phone number column? We use `ALTER` to carefully modify the blueprint without destroying the table.

```
-- Adding a new column to an existing table
ALTER TABLE student ADD phone_number varchar(15);

-- Changing the data type of an existing column
ALTER TABLE student ALTER COLUMN name varchar(100);
```

5.7.3 3. The `DROP` Command

When a table is completely obsolete, we use `DROP`.

```
DROP TABLE student;
```

> **Warning:** Use `DROP` very carefully! It wipes out the table structure and everything inside it instantly.

6 Data Manipulation (DML) & Basic Queries

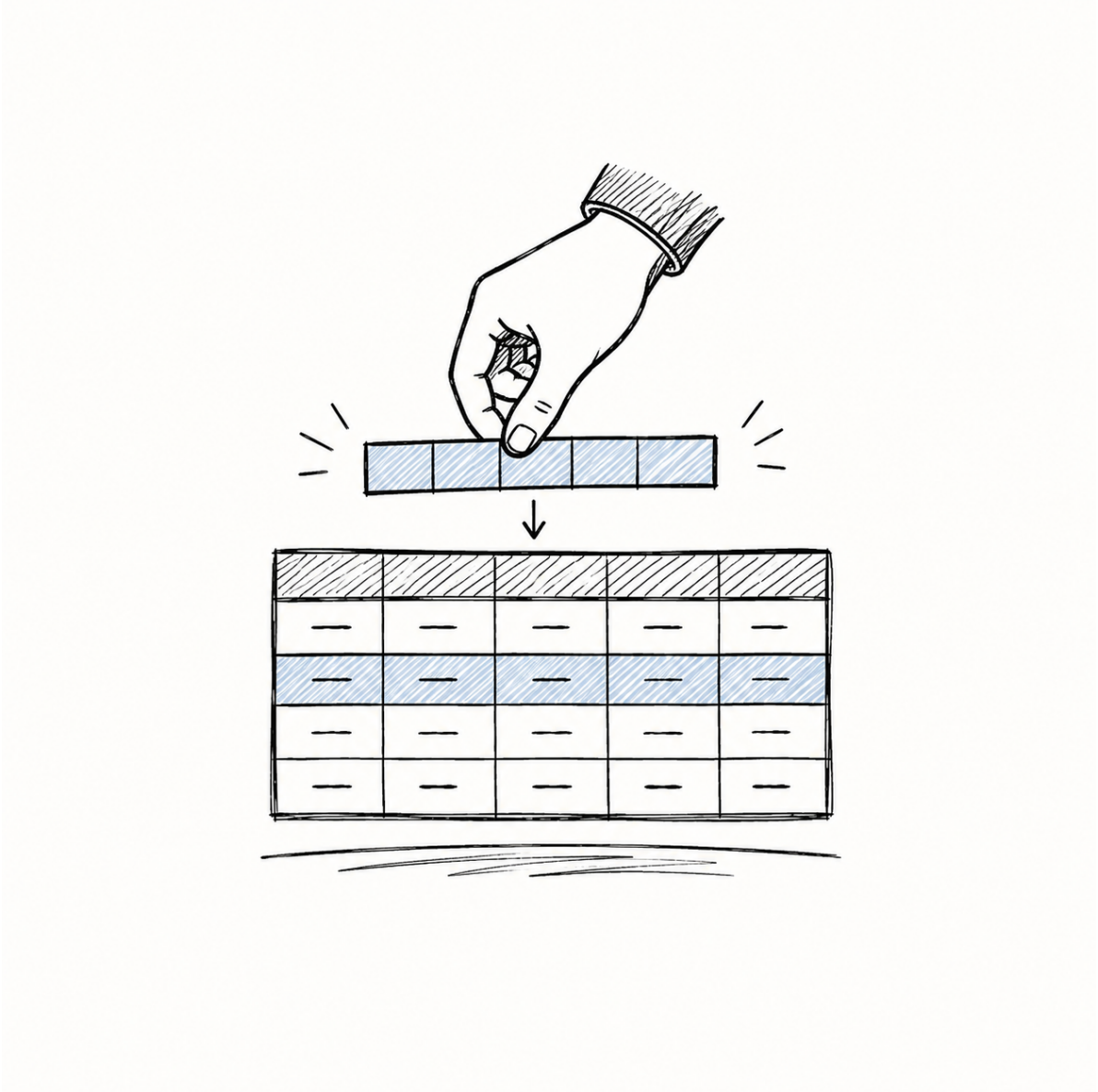


Figure 6.1: DML Operations

6.1 DML: Furnishing the House

In the previous section, DDL acted as our architect—building the empty structure of our `student` table. Now, it's time for **DML (Data Manipulation Language)** to move the furniture in! DML handles all the inserting, modifying, and deleting of the *actual data rows*.

Let's start with an empty table:

roll_no	name	level	course	term	grade
(empty)					

6.1.1 1. `INSERT`: Adding New Records

A new student, Pravin, has just enrolled. We need to add him to our database.

```
INSERT INTO student (roll_no, name, level, course, term, grade)
VALUES ('26f10', 'Pravin', 'Degree', 'DBMS', '27t1', 'S');
```

> **Tip:** You *can* write an `INSERT` without listing the column names (e.g., `INSERT INTO student VALUES (...)`), but specifying the columns as shown above is much safer! It prevents errors if someone later adds a new column to the table.

Inserting Multiple Rows at Once: You can also insert several students in a single command by separating the values with commas!

```
INSERT INTO student (roll_no, name, level, course, term, grade)
VALUES
    ('26f11', 'Alice', 'Degree', 'Math', '27t1', 'B'),
    ('26f12', 'Bob', 'Diploma', 'Physics', '27t1', 'C');
```

Our Table Now:

roll_no	name	level	course	term	grade
26f10	Pravin	Degree	DBMS	27t1	S

6.1.2 2. `UPDATE`: Modifying Existing Data

Oops! We made a mistake. Pravin actually got an 'A' grade, not an 'S'. We don't delete his record; we just `UPDATE` it.

```
UPDATE student
SET grade = 'A'
WHERE roll_no = '26f10';
```

> [**WARNING**] Always, *always* use a `WHERE` clause with `UPDATE`!

What happens if you forget the `WHERE` clause? If you simply ran `UPDATE student SET grade = 'A';`, it would ignore the individual student and change the grade of **every single student** in the entire database to an 'A'!

roll_no	name	grade
26f10	Pravin	**A**
26f11	Alice	**A**
26f12	Bob	**A**

(Oops! Alice and Bob just got free 'A' grades because you forgot the WHERE clause!)

Our Table After Correct Update:

roll_no	name	level	course	term	grade
26f10	Pravin	Degree	DBMS	27t1	**A**

6.1.3 3. `DELETE`: Removing Records

Pravin has decided to transfer to another school. We need to remove his record completely.

```
DELETE FROM student
WHERE roll_no = '26f10';
```

(Just like `UPDATE`, forgetting the `WHERE` clause here would delete everyone!)

6.2 The Anatomy of a Query

Now that we have data, how do we ask the database questions? We write queries.

Think of a SQL query like giving instructions to an incredibly fast, but very literal librarian. The fundamental structure requires three clauses:

```
SELECT <attributes>
FROM <tables>
WHERE <condition>;
```

Here is the mental model of how the librarian thinks: 1. **FROM**: *"Which section of the library (tables) should I go to?"* 2. **WHERE**: *"Which specific books (rows) should I pull off the shelf?"* 3. **SELECT**: *"Which chapters (columns) should I photocopy and give to you?"*

6.2.1 The SELECT Clause in Detail

The SELECT clause dictates what the final output looks like.

- * : "Give me everything." (All columns)
- DISTINCT : "Remove any duplicate answers."
- AS : "Rename this column in the output just to make it look nicer."
- TOP <n> (or LIMIT): "Only show me the first n results."

Example:

```
SELECT DISTINCT(name) AS Instructor_Name
FROM instructor
WHERE dept_name = 'Biology';
```

6.2.2 The ORDER BY Clause

By default, SQL returns rows in whatever internal order it feels like. If you want them sorted, you must explicitly use the ORDER BY clause at the very end of your query.

- ASC: Ascending order (A-Z, 1-10). This is the default.
- DESC: Descending order (Z-A, 10-1).

Example:

```sql
-- Sort instructors by salary from highest to lowest
SELECT name, salary
FROM instructor
ORDER BY salary DESC;
```

6.3 The Danger of Cross Joins

What happens if you tell the librarian to get data from *two* places at once?

```sql
SELECT *
FROM instructor, course;
```

When you list multiple tables in the `FROM` clause without a `WHERE` condition, SQL performs a **Cross Join** (Cartesian Product). It pairs *every* row in the first table with *every* row in the second table.

Let's visualize this: If you have 2 Instructors and 2 Courses:

Instructors Table:

id	name
1	Alice
2	Bob

Courses Table:

c_id	title
101	Math
102	Science

The Cross Join Result:

id	name	c_id	title
1	Alice	101	Math
1	Alice	102	Science
2	Bob	101	Math

id	name	c_id	title
2	Bob	102	Science

(2 instructors × 2 courses = 4 rows!)

Imagine if you had 1,000 instructors and 1,000 courses—you'd instantly generate **1 million rows**! Most of these combinations are nonsense (Alice doesn't teach every single course).

6.3.1 Fixing the Cross Join

To fix this, we use the `WHERE` clause to filter the massive Cross Join down to only the combinations that make sense (e.g., linking the instructor to the specific course they actually teach):

```
SELECT *
FROM instructor AS i, course AS c
WHERE i.course_id = c.course_id
  AND i.dept_name = 'Biology';
```

(Notice how we used `AS i` and `AS c` to give our tables nicknames? This makes our `WHERE` clause much easier to read!)

7 Advanced SQL Filtering, Sets & Aggregation



Figure 7.1: Advanced SQL Filtering

7.1 Advanced Filtering

Sometimes, a simple `=` or `>` isn't enough. What if you only remember that an instructor's name *starts with* an 'M', or you need to check if their salary falls within a specific range?

7.1.1 1. String Operations (`LIKE`)

`LIKE` is SQL's built-in detective. It performs pattern matching on text using two special wildcard characters:

- `%` : Matches **any sequence** of characters (even zero characters).
- `_` : Matches exactly **one** character.

Examples in Action:

- `LIKE 'Intro%'` : Matches any string starting with "Intro" (e.g., "Introduction", "Intro to CS").
- `LIKE '%Comp%'` : Matches any string containing "Comp" anywhere inside it (e.g., "Computer", "MyComp").
- `LIKE '___'` (three underscores) : Matches any string of exactly three characters (e.g., "Cat", "Dog").

```
SELECT name
FROM instructor
WHERE name LIKE '%dar%';
-- Finds names like 'Mandar', 'Darren', 'Cedar'
```

7.1.2 2. Advanced Predicates (`BETWEEN` and `IN`)

These are essentially shortcuts (syntax sugar) that save you from writing long, messy `WHERE` clauses.

`BETWEEN a AND b`: A clean way to check for a range (inclusive of both `a` and `b`).

```
-- Instead of: WHERE salary >= 90000 AND salary <= 100000
SELECT name
FROM instructor
WHERE salary BETWEEN 90000 AND 100000;
```

`IN`: A shorthand for multiple `OR` conditions.

```sql
-- Instead of: WHERE dept_name = 'Comp Sci' OR dept_name = 'Biology'
SELECT name
FROM instructor
WHERE dept_name IN ('Comp Sci', 'Biology');
```

7.2 Dealing with the Unknown: `NULL`

In databases, what happens when we simply *don't know* a value? For example, a new student hasn't been assigned a dorm yet. We don't enter "0" or "None" because those are actual values. Instead, we use `NULL`.

`NULL` signifies a missing or unknown value. It is **not** empty text, and it is **not** the number zero.

7.2.1 The `IS` Operator

Because `NULL` means "unknown", we cannot use standard mathematical operators like `=` or `!=` on it.

If you ask SQL: `WHERE dorm = NULL`, the result is practically "Unknown = Unknown?", which doesn't make sense! Instead, we must use the special `IS` operator.

```sql
-- Find students who haven't been assigned a dorm yet
SELECT name
FROM student
WHERE dorm IS NULL;

-- Find students who HAVE a dorm
SELECT name
FROM student
WHERE dorm IS NOT NULL;
```

7.2.2 Three-Valued Logic

Because of `NULL`, SQL doesn't just use standard boolean logic (True / False). It uses **Three-Valued Logic**:

1. **True**

2. **False**
3. **Unknown** (The result of comparing something to `NULL`)

Here's how `Unknown` impacts your queries:

- `True AND Unknown` → `Unknown`
- `False AND Unknown` → `False` (Because False ruins everything in an AND)
- `True OR Unknown` → `True` (Because True saves everything in an OR)
- `False OR Unknown` → `Unknown`

  Critical Rule: The `WHERE` clause will *only* return rows where the final condition evaluates to exactly **True**. If the condition evaluates to **False** or **Unknown**, the row is thrown out!

7.3 Set Operations

Because relational databases are built on mathematical Set Theory, we can perform operations on the results of two different queries!

Imagine Query A returns `[Alice, Bob, Charlie]` and Query B returns `[Bob, Charlie, David]`.

Operator	Action	Result
UNION	Combines both sets, removing duplicates.	`[Alice, Bob, Charlie, David]`
INTERSECT	Keeps only what is present in *both* sets.	`[Bob, Charlie]`
EXCEPT	Keeps what is in Set A, but *subtracts* anything found in Set B.	`[Alice]`

Note: By default, these operations remove duplicate rows. If you want to keep duplicates, add the `ALL` keyword (e.g., `UNION ALL`).

7.4 Aggregation & Grouping

Often, we don't care about the individual rows. If a university has 50,000 students, we might just want to know the *total* number of students or the *average* GPA.

Aggregate functions take a whole column of data, mash it together, and return a single summary value.

- `avg()`: Average value
- `min()`: Minimum value
- `max()`: Maximum value
- `sum()`: Sum of all values
- `count()`: Count of rows

7.4.1 `GROUP BY`: Sorting into Buckets

What if we want the average salary *for every single department*? We can't just use `avg(salary)` alone, because that gives us the average of the entire university.

Instead, we use `GROUP BY`. This tells SQL: "Sort the rows into buckets based on their department. Then, calculate the average for each bucket independently!"

```
SELECT dept_name, avg(salary)
FROM instructor
GROUP BY dept_name;
```

7.4.2 `HAVING`: The Bouncer for Buckets

We already learned that `WHERE` filters individual rows. But what if we want to filter the *groups* we just created? For example, "Show me departments where the *average salary* is greater than 42,000."

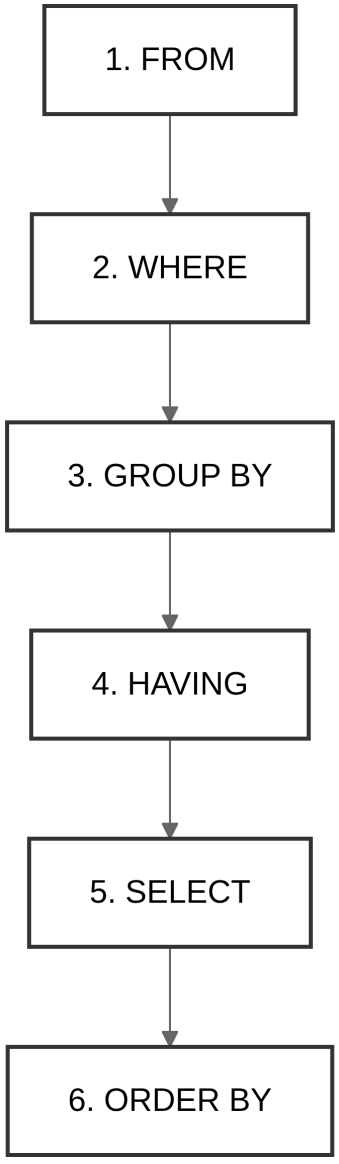
You cannot use `WHERE avg(salary) > 42000` because `WHERE` happens *before* the buckets are created!

Instead, we use `HAVING`. If `WHERE` is the bouncer for rows, `HAVING` is the bouncer for groups!

```
SELECT dept_name, avg(salary)
FROM instructor
GROUP BY dept_name
HAVING avg(salary) > 42000;
```

7.5 The SQL Execution Order

When you write a complex SQL query, the database does *not* read it from top to bottom! It processes the clauses in a very specific order. Understanding this order is the ultimate secret to mastering SQL.



Step-by-Step Narrative:

1. `FROM`: The engine goes to the tables to gather the raw data.
2. `WHERE`: It filters out the individual rows that don't meet the criteria.
3. `GROUP BY`: It takes the surviving rows and sorts them into buckets.
4. `HAVING`: It looks at the buckets, calculates the aggregates, and throws away any buckets that fail the test.
5. `SELECT`: It finally decides which columns to show you on the screen.
6. `ORDER BY`: It alphabetizes or sorts the final output for you.

Part III

Week-03

8 Nested Subqueries

8.1 Nested Subqueries

A **subquery** is simply a `select-from-where` block placed *inside* another query. SQL lets you nest subqueries in the `WHERE`, `FROM`, and `SELECT` clauses.

- `select` : Must return a single scalar value.
- `from` : Can return any valid relation (table).
- `where` : Used in expressions like `Attribute <operation> (subquery)`.

8.2 Subqueries in the `WHERE` Clause

Typically used to test **set membership**, **set comparison**, and **set cardinality**.

8.2.1 Set Membership: `IN` / `NOT IN`

Example: Find employees who worked on Project A **and** Project B.

```
select distinct emp_id
from assignment
where project = 'Project A' and
      emp_id in (select emp_id
                 from assignment
                 where project = 'Project B');
```

Swap `in` for `not in` to find employees on Project A **but not** Project B.

8.2.2 Set Comparison: `SOME` and `ALL`

Clause	Meaning	Logic
`some`	True if comparison holds for **at least one** row in subquery	$\exists\, t \in r : F\langle \text{comp} \rangle\, t$
`all`	True if comparison holds for **every** row in subquery	$\forall\, t \in r : F\langle \text{comp} \rangle\, t$

<comp> can be: $<$, $\leq$, $>$, $\geq$, $=$,

Example: Employees earning more than **some** employee in Sales:

```
select name
from employee
where salary > some (select salary
                     from employee
                     where dept_name = 'Sales');
```

Example: Employees earning more than **all** employees in Sales:

```
select name
from employee
where salary > all (select salary
                    from employee
                    where dept_name = 'Sales');
```

8.2.3 Test for Empty Sets: `EXISTS`

- `exists r` is true if `r` is non-empty.
- `not exists r` is true if `r` is empty.

Example: Find projects active in both Q1 2025 and Q2 2025, using a **correlated subquery** (inner query refers to the outer query's variable).

```
select project_id
from timeline as S
where quarter = 'Q1' and year = 2025 and
      exists (select *
              from timeline as T
              where quarter = 'Q2' and year = 2025
                    and S.project_id = T.project_id);
```

- `S` is the **correlation name** (outer query variable)

- The inner query referencing `S` is the **correlated subquery**

Example: Find employees who have worked on **all** projects in the Marketing department (uses `not exists` + set difference logic):

```
select distinct S.emp_id, S.name
from employee as S
where not exists (
    (select project_id from project where dept_name = 'Marketing')
    except
    (select T.project_id from assignment as T where S.emp_id = T.emp_id)
);
```

8.2.4 Test for Duplicates: `UNIQUE`

`unique (subquery)` is true if the subquery's result has **no duplicate rows**.

```
select T.project_id
from project as T
where unique (select R.project_id
              from assignment as R
              where T.project_id = R.project_id and R.year = 2025);
```

(Finds projects staffed by at most one employee in 2025.)

8.3 Subqueries in the FROM Clause

A subquery can stand in for a table, letting you filter on an **aggregate** without `HAVING`.

Example: Average salary of departments where average salary exceeds $60,000:

```
select dept_name, avg_salary
from (select dept_name, avg(salary) as avg_salary
      from employee
      group by dept_name) as dept_avg
where avg_salary > 60000;
```

8.3.1 The `WITH` Clause

`with` defines a **temporary, named result** (like a scratchpad table) visible only to the query that follows.

Example: Find the name of the department that has the maximum budget.

```
with max_budget(value) as
    (select max(budget) from department)
select department.name
from department, max_budget
where department.budget = max_budget.value;
```

Multiple `with` blocks can be chained for multi-step logic:

Example: Find the names of the departments whose total salary expense is greater than the average total salary expense across all departments.

```
with dept_total(dept_name, value) as
    (select dept_name, sum(salary)
     from employee
     group by dept_name),
dept_total_avg(value) as
    (select avg(value) from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

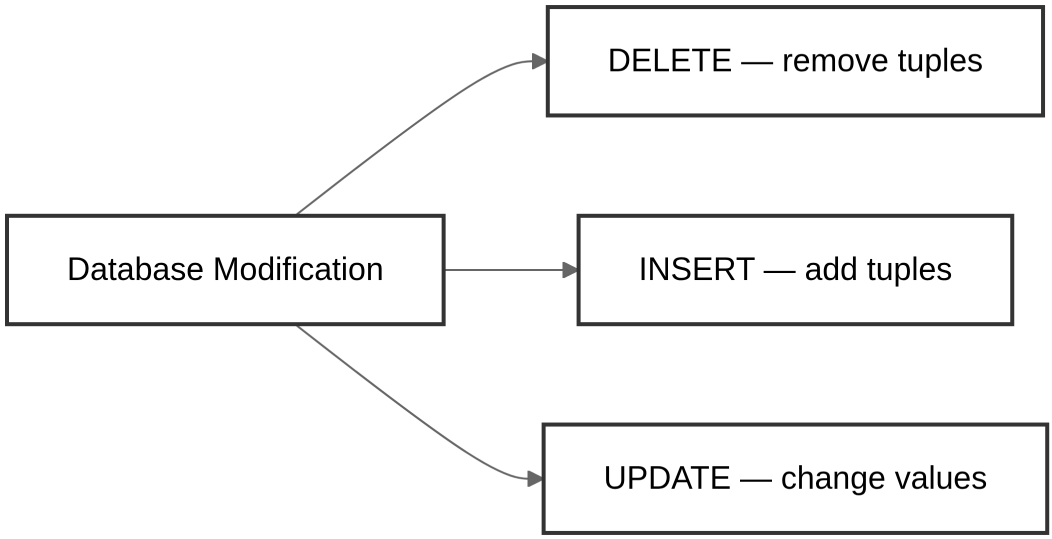
8.4 Subqueries in the SELECT Clause

A **scalar subquery** returns exactly one value, used wherever a single value is expected.

```
select dept_name,
       (select count(*)
        from employee
        where department.dept_name = employee.dept_name) as num_employees
from department;
```

If the subquery returns more than one row, this causes a **runtime error**.

8.5 Modifying the Database



8.5.1 Deletion

```
-- Delete all employees
delete from employee;

-- Delete employees from one department
delete from employee where dept_name = 'Finance';

-- Delete employees in departments located in a specific building
delete from employee
where dept_name in (select dept_name
                    from department
                    where building = 'Tower B');
```

Tricky case: Delete employees earning less than the average salary.

```
delete from employee
where salary < (select avg(salary) from employee);
```

> As rows are deleted, the average changes! Solution: **(1)** computing the average and identifying all rows to delete *first*, then **(2)** deleting them — without recomputing mid-way.

8.5.2 Insertion

```
insert into course
values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);

-- with tot_creds set to null
insert into student
values ('3003', 'Green', 'Finance', null);

-- insert from another query's results
insert into student
select id, name, dept_name, 0
from employee;
```

> The entire `select-from-where` is evaluated **before** any row is inserted — otherwise `insert into table1 select * from table1` would loop infinitely.

8.5.3 Updates

Conditional updates — give a 3% raise above \$100K, 5% below (order matters!):

```
update employee set salary = salary * 1.03 where salary > 100000;
update employee set salary = salary * 1.05 where salary <= 100000;
```

Cleaner with a **case** statement (single pass, no ordering issues):

```
update employee
set salary = case
    when salary <= 100000 then salary * 1.05
    else salary * 1.03
end;
```

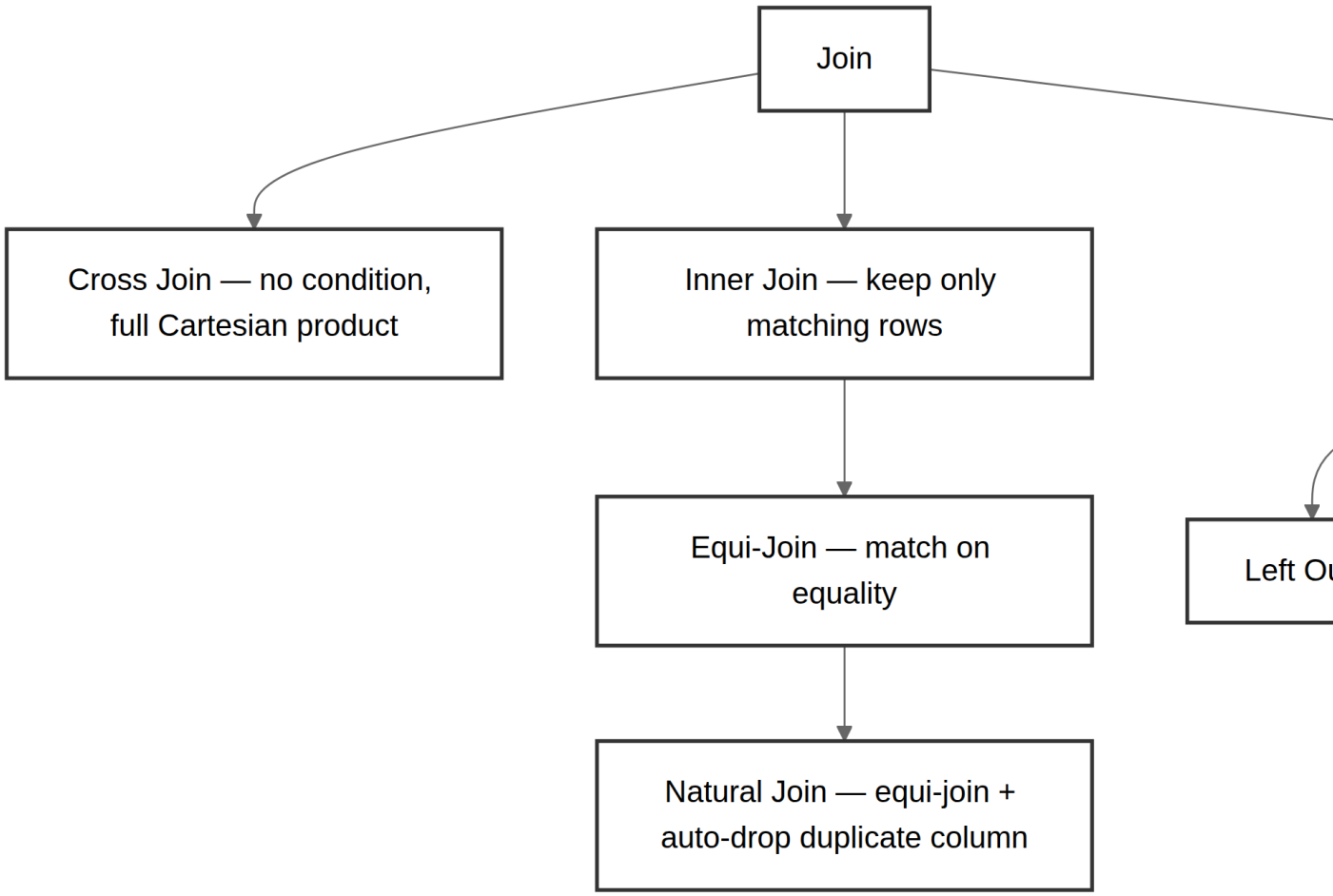
Updates with scalar subqueries — recompute total credits per student:

```
update student S
set tot_creds = (select sum(credits)
                 from takes, course
                 where takes.course_id = course.course_id and
                       S.id = takes.id and
                       takes.grade <> 'F' and
                       takes.grade is not null);
```

(Find the total credits earned by each student (excluding failed or uncollected grades) and update their record in the student table.)

9 Joins

A **join** takes two tables and stitches them into one, based on some matching condition. Think of it as a Cartesian product (every row paired with every row) that's then *filtered* down to only the pairs that actually make sense together.



9.1 Sample Data

To see the difference between join types, imagine two small tables:

movie

movie_id	title
M1	Inception
M2	Arrival
M3	Dune

review

movie_id	rating
M1	9
M2	8
M4	7

Notice: `Dune (M3)` has no review, and the review for `M4` doesn't match any movie. This mismatch is exactly what different join types handle differently.

9.2 Cross Join

Returns the **full Cartesian product** — every row of one table paired with every row of the other, no filtering at all.

```
-- Explicit
select * from movie cross join review;

-- Implicit (comma syntax)
select * from movie, review;
```

With 3 movies and 3 reviews, this produces **9 rows** — almost all of them meaningless pairings.

9.3 Inner Join

Keeps only the rows where the join condition is satisfied on **both sides**.

Join Type	Condition	Result
Equi-Join	Explicit equality condition (`on a.x = b.x`)	Both `x` columns kept
Natural Join	Equality assumed on same-named columns	Duplicate column auto-dropped

```
-- Equi-join
select * from movie inner join review
on movie.movie_id = review.movie_id;

-- Natural join (same result, but movie_id appears once)
select * from movie natural join review;
```

Result (only matching `movie_id`s survive):

movie_id	title	rating
M1	Inception	9
M2	Arrival	8

`M3` (no review) and `M4` (no movie) both **disappear** — this is the information loss outer joins solve.

9.4 Outer Join

Outer joins keep the inner join result **plus** the unmatched rows from one or both sides, filling missing columns with `null`.

9.4.1 Left Outer Join

Keeps **all rows from the left table**, even unmatched ones.

```
select * from movie natural left outer join review;
```

movie_id	title	rating
M1	Inception	9
M2	Arrival	8
M3	Dune	null

9.4.2 Right Outer Join

Keeps **all rows from the right table**, even unmatched ones.

```
select * from movie natural right outer join review;
```

movie_id	title	rating
M1	Inception	9
M2	Arrival	8
M4	null	7

9.4.3 Full Outer Join

Keeps **everything from both sides** — the union of left and right outer joins.

```
select * from movie natural full outer join review;
```

movie_id	title	rating
M1	Inception	9
M2	Arrival	8
M3	Dune	null
M4	null	7

9.4.4 Using Explicit Conditions

Joins can also use `on` (any condition) or `using` (shared column names), instead of `natural`:

```sql
-- Explicit condition
select * from movie left outer join review
on movie.movie_id = review.movie_id;

-- Shared column shorthand
select * from movie full outer join review
using (movie_id);
```

`on` works with *any* condition (even inequality!). `natural` and `using` both rely on matching column names — `using` lets you name exactly which shared column(s) to join on, while `natural` matches on **all** identically-named columns.

9.5 Quick Reference: Join Types

Join	Unmatched left rows kept?	Unmatched right rows kept?
Cross Join	n/a (no matching)	n/a
Inner Join		
Left Outer		
Right Outer		
Full Outer		

---

10 Intermediate SQL

10.1 Views

10.1.1 Why Views?

Not everyone should see the full picture. Suppose someone needs an employee's name and department, but **not** their salary. Instead of trusting every query to remember to hide the salary column, define it once:

```
select id, name, dept_name from employee;
```

A **view** is exactly this — a saved query that behaves like a "virtual table." It's not real stored data; it's a name attached to a query expression that gets substituted in wherever the view is used.

10.1.2 Creating a View

```
create view v as <query expression>;
```

```
-- A view of employees without salary
create view staff_directory as
select id, name, dept_name from employee;

-- Use it like a table
select name from staff_directory where dept_name = 'Marketing';

-- A view with computed columns
create view dept_salary_totals(dept_name, total_salary) as
select dept_name, sum(salary)
from employee
group by dept_name;
```

10.1.3 Views Built on Views

Views can refer other views, not just base tables:

```
create view marketing_2025_projects as
select project.project_id, lead_emp_id, building, room_number
from project, assignment
where project.project_id = assignment.project_id
  and project.dept_name = 'Marketing'
  and assignment.year = 2025;

create view marketing_2025_tower_b as
select project_id, room_number
from marketing_2025_projects
where building = 'Tower B';
```

- `v1` **depends directly on** `v2` if `v2` appears in `v1`'s definition.
- `v1` **depends on** `v2` if there's any chain of dependency leading from `v1` to `v2`.
- A view is **recursive** if it depends on itself.

10.1.4 View Expansion

To understand what a nested view *really* means, SQL conceptually **expands** it: repeatedly replace each view name with its underlying query, until only base tables remain.

```
repeat:
    find a view reference v_i in the expression
    replace v_i with v_i's defining query
until no view references remain
```

This terminates as long as no view is recursive.

10.1.5 Updating Views

Inserting into a view secretly inserts into the underlying table:

```
insert into staff_directory values ('E501', 'Lee', 'Design');
-- Translates to:
-- insert into employee values ('E501', 'Lee', 'Design', null);
```

But this only works cleanly for *simple* views — most databases require:

- `from` clause has exactly **one** base table
- `select` lists plain column names only (no expressions, aggregates, or `distinct`)
- Any column left out of the `select` can default to `null`
- No `group by` or `having`

If a view joins multiple tables, an insert can be **ambiguous** (e.g., which department's row gets the new value if several departments share a building?) — and some inserts that satisfy the view's *output* can violate its `where` clause entirely, making the update meaningless to translate.

10.1.6 Materialized Views

A **materialized view** physically stores the view's result as a real table, instead of recomputing it on every query — faster to read, but it goes **stale** the moment the underlying tables change. Keeping it correct requires actively **maintaining** it (refreshing it) whenever the source data updates.

View Type	Storage	Speed	Freshness
Regular View	None (just a saved query)	Recomputed every time	Always current
Materialized View	Physical table	Fast to read	Needs manual/scheduled refresh

10.2 Transactions

A **transaction** is a unit of work — a group of one or more SQL statements that should succeed or fail *as a whole*.

- **Atomic**: either the *entire* transaction completes, or none of it does (rolled back as if it never happened).
- **Isolated**: a transaction shouldn't see the messy, half-finished state of another transaction running at the same time.
- Transactions begin **implicitly**, and end with either:
  - `commit work` — make all changes permanent
  - `rollback work` — undo everything since the transaction began

10.3 Integrity Constraints

Constraints stop *valid-looking but wrong* changes from silently corrupting the database — e.g., "an employee's salary must be at least $15/hour" or "every customer must have a phone number."

10.3.1 On a Single Relation

Constraint	Purpose
`not null`	Column can never be empty
`primary key`	Uniquely identifies each row; implies `not null` + `unique`
`unique (A1, ..., Am)`	Columns form a **candidate key** — but unlike primary keys, candidate keys *can* be null
`check(P)`	Row must satisfy predicate `P`

```
-- not null
name varchar(20) not null,
budget numeric(12,2) not null

-- check: restrict allowed values
create table project (
    project_id varchar(8),
    status varchar(10),
    primary key (project_id),
    check (status in ('Active', 'On Hold', 'Completed', 'Cancelled'))
);
```

10.4 Referential Integrity

Ensures a value referenced in one table **actually exists** in the table it points to. If `'Design'` appears as a department name in `employee`, there must be a matching row for `'Design'` in `department`.

> **Formal definition:** If `A` is the primary key of relation `S`, then `A` is a **foreign key** of relation `R` if every value of `A` appearing in `R` also appears in `S`.

10.4.1 Cascading Actions

What should happen when a referenced row is deleted or updated?

```
create table project (
    project_id char(5) primary key,
    title varchar(20),
    dept_name varchar(20) references department
);

-- with explicit cascading behavior
create table project (
    ...
    dept_name varchar(20),
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    ...
);
```

Action	Effect when referenced row is deleted/updated
`cascade`	Automatically delete/update matching rows too
`no action`	Block the delete/update if references exist (default)
`set null`	Set the foreign key to `null`
`set default`	Set the foreign key to its default value

10.4.2 The Chicken-and-Egg Problem

Self-referencing constraints can deadlock a single insert. Consider:

```
create table employee (
    id char(10),
    name char(40),
    manager_id char(10),
    primary key (id),
    foreign key (manager_id) references employee
);
```

How do you insert the very first employee if their manager must already exist? Three options:

1. Insert the manager *before* the employee who reports to them.
2. Set `manager_id` to `null` initially, then `update` it later (impossible if the column is `not null`).
3. **Defer** constraint checking until the transaction commits (covered in advanced topics).

10.5 SQL Data Types and Schemas

10.5.1 Built-in Date/Time Types

Type	Stores	Example
`date`	Year, month, day	`date '2025-7-27'`
`time`	Hours, minutes, seconds	`time '09:00:30'`
`timestamp`	Date + time together	`timestamp '2025-7-27 09:00:30.75'`
`interval`	A span of time	`interval '1' day`

Subtracting two dates/times gives you an `interval`; adding an `interval` to a date/time shifts it forward or backward.

10.5.2 Indexes

An **index** is a shortcut structure that lets the database jump straight to matching rows instead of scanning the whole table.

```
create table student (
    id varchar(5),
    name varchar(20) not null,
    dept_name varchar(20),
    tot_cred numeric(3,0) default 0,
    primary key (id)
);

create index student_id_index on student(id);
```

```
select * from student where id = '12345';
-- with the index above, this skips a full table scan
```

10.5.3 User-Defined Types (UDT)

Like a `typedef` in C — gives a meaningful name to an existing type.

```
create type dollars as numeric(12,2) final;

create table department (
    dept_name varchar(20),
    building varchar(15),
    budget dollars
);
```

10.5.4 Domains

Similar to UDTs, but domains can carry their own **constraints**.

```
create domain person_name char(20) not null;

create domain degree_level varchar(10)
    constraint degree_level_test
    check (value in ('Bachelors', 'Masters', 'Doctorate'));
```

10.5.5 Large Object Types

For storing big, unstructured content like photos, videos, or files:

Type	Stores
`blob`	**B**inary **L**arge **OB**ject — raw binary data
`clob`	**C**haracter **L**arge **OB**ject — large character data

A query returns a **pointer** to a large object, not the object itself — the actual data is fetched separately.

10.6 Authorization

10.6.1 Forms of Authorization

On data:

Privilege	Allows
`read`	Viewing data only
`insert`	Adding new rows
`update`	Modifying existing rows
`delete`	Removing rows

On schema:

Privilege	Allows
`index`	Creating/dropping indices
`resources`	Creating new relations
`alteration`	Adding/removing columns
`drop`	Deleting relations entirely

10.6.2 Granting Privileges

```
grant <privilege list> on <relation or view> to <user list>;
```

`<user list>` can be a specific user, `public` (everyone), or a **role**.

```
grant select on employee to U1, U2, U3;
grant insert, update on project to U1;
grant all privileges on department to admin_user;
```

> The grantor must already **hold** the privilege themselves (or be the DBA). Granting a privilege on a *view* does **not** automatically grant privileges on the tables underneath it.

10.6.3 Revoking Privileges

```
revoke <privilege list> on <relation or view> from <user list>;
```

```
revoke select on department from U1, U2, U3;
```

- `revoke all from ...` removes every privilege the user holds.
- Revoking from `public` removes the privilege from everyone **except** users who were granted it explicitly elsewhere.
- If the same privilege was granted by **two different** grantors, revoking from one grantor may leave the privilege intact (granted by the other).
- Revoking a privilege also revokes any privileges that **depended** on it.

10.6.4 Roles

Roles bundle privileges so you don't have to grant the same set repeatedly.

```
create role instructor;
grant select on takes to instructor;
grant instructor to amit;       -- amit inherits the privileges of 'instructor'

-- Roles can inherit from other roles
create role teaching_assistant;
grant teaching_assistant to instructor;   -- instructor now also gets TA's privileges

-- Chains of roles
create role dean;
grant instructor to dean;
grant dean to satoshi;          -- satoshi inherits dean → instructor → TA
```

10.6.5 Authorization on Views

```
create view geo_employees as
    (select * from employee where dept_name = 'Geology');

grant select on geo_employees to geo_staff;
```

If `geo_staff` queries `geo_employees`, it works even if `geo_staff` has **no** direct permission on the underlying `employee` table — the view's permissions are what matter, not the creator's original access (though if the view's *creator* lacked permission on `employee`, the view itself couldn't have been created correctly in the first place).

10.6.6 Other Authorization Features

```
-- 'references' privilege needed to create a foreign key pointing at someone else's table
grant references (dept_name) on department to mariano;

-- Transferable privileges
grant select on department to amit with grant option;

-- Revoking with dependency handling
revoke select on department from amit, satoshi cascade;   -- also revokes anything amit/satos
revoke select on department from amit, satoshi restrict;  -- blocks the revoke if dependents
```

The `references` privilege exists because a foreign key constraint can restrict what the *referenced* table's owner is allowed to do (e.g., blocking a delete due to `on delete no action`) — so creating that link requires explicit permission.

11 Advanced SQL

11.1 Why Go Beyond Plain SQL?

SQL is **declarative** — great for describing *what* you want, clumsy for *how* to compute it step by step. Sometimes you need loops, conditionals, and reusable logic. SQL:1999 added **functions, procedures, and control-flow constructs** to bridge that gap, turning SQL into something closer to a general-purpose language.

11.2 SQL Functions

A **function** returns a single value, and can be written in SQL itself — or in an external language (useful for specialized types like images or geometry, e.g. "do these two polygons overlap?").

```
create function emp_count(dname varchar(20))
returns integer
begin
    declare e_count integer;
    select count(*) into e_count
    from employee
    where employee.dept_name = dname;
    return e_count;
end
```

Use it just like any expression:

```
select dept_name, budget
from department
where emp_count(dept_name) > 12;
```

Keyword	Meaning
`begin ... end`	A **compound statement** — holds multiple SQL statements together
`returns`	Declares the *type* of the result (e.g., `integer`)
`return`	Specifies the *actual value* sent back when the function finishes

> Think of SQL functions as **parameterized views** — a view that generalizes by accepting arguments.

11.2.1 Table Functions

Added in SQL:2003 — instead of returning one value, a table function returns an **entire relation**.

```
create function employees_in(dname varchar(20))
returns table (
    id varchar(5),
    name varchar(20),
    dept_name varchar(20),
    salary numeric(8,2)
)
return table
    (select id, name, dept_name, salary
     from employee
     where employee.dept_name = employees_in.dname);
```

```
select * from table(employees_in('Marketing'));
```

11.2.2 Procedures

Where a function returns a value via `return`, a **procedure** passes results back through `out` parameters.

```
create procedure emp_count_proc(
    in dname varchar(20),
    out e_count integer)
begin
    select count(*) into e_count
```

```
    from employee
    where employee.dept_name = emp_count_proc.dname;
end
```

```
declare e_count integer;
call emp_count_proc('Design', e_count);
```

SQL:1999 allows **overloading** — multiple functions/procedures can share a name as long as their parameter count or types differ.

11.3 Procedural Language Constructs

Most databases implement their own dialect of these — syntax varies across systems.

11.3.1 Loops

```
-- while: check condition before each iteration
while boolean_expression do
    sequence_of_statements;
end while;

-- repeat: check condition after each iteration (always runs once)
repeat
    sequence_of_statements;
until boolean_expression
end repeat;

-- for: iterate directly over query results
declare total integer default 0;
for r as
    select budget from department
do
    set total = total + r.budget;
end for;
```

11.3.2 Conditionals

```
-- if / elseif / else
if boolean_expression then
    sequence_of_statements;
elseif boolean_expression then
    sequence_of_statements;
else
    sequence_of_statements;
end if;
```

```
-- simple case: match a variable against fixed values
case dept_name
    when 'Sales' then ...
    when 'Design' then ...
    else ...
end case;

-- searched case: match arbitrary expressions
case
    when salary > 100000 then ...
    when salary > 50000 then ...
    else ...
end case;
```

11.3.3 Exception Handling

```
declare seat_unavailable condition;
declare exit handler for seat_unavailable
begin
    ...
    signal seat_unavailable;
    ...
end
```

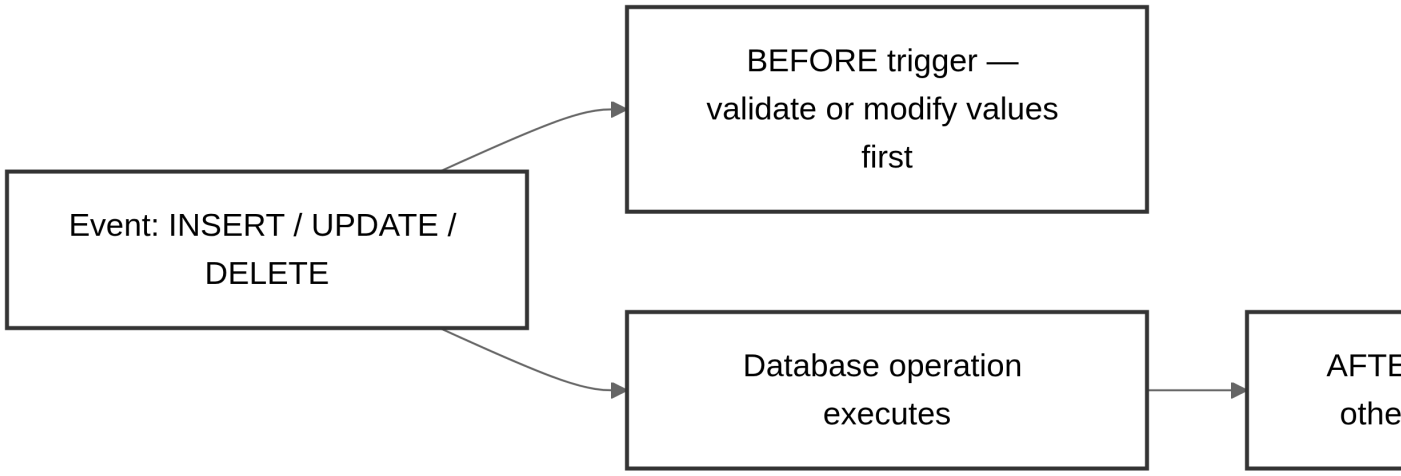
The `exit` handler terminates the enclosing `begin...end` block when the condition fires. Other handler behaviors are also possible (e.g., continue instead of exit).

11.4 Triggers

A **trigger** is code that fires automatically in response to an `insert`, `update`, or `delete` on a table — no one has to remember to call it.

Defining a trigger means specifying three things:

1. **Event** — insert / update / delete
2. **Timing** — `BEFORE` or `AFTER` the event
3. **Action** — what to actually do



11.4.1 BEFORE vs. AFTER

Timing	Typical use
BEFORE	Clean up or validate incoming values before they're written (e.g., normalize blanks to `null`); raise errors before a delete proceeds
AFTER	Update *other* tables to stay in sync, maintain audit trails, enforce cross-table integrity, trigger alerts/notifications

11.4.2 Row-Level vs. Statement-Level

Type	Behavior
Row-level	Fires once **per affected row**. A 10% raise across 100 employees fires the trigger 100 times.
Statement-level	Fires **once per statement**, regardless of how many rows were touched — uses `referencing old table` / `referencing new table` to access all affected rows at once via transition tables. More efficient for bulk updates.

11.4.3 Example: BEFORE trigger (data cleanup)

Convert blank grades to `null` before they're written:

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
    set nrow.grade = null;
end;
```

11.4.4 Example: AFTER trigger (cross-table maintenance)

Update a student's earned credits whenever a failing/blank grade turns into a pass:

```
create trigger credits_earned after update of grade on (takes)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
     and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred = tot_cred +
        (select credits from course where course.course_id = nrow.course_id)
    where student.id = nrow.id;
end;
```

- `referencing old row as` — access pre-update values (deletes & updates)

- `referencing new row as` — access post-update values (inserts & updates)
- Triggers can be scoped to specific columns: `after update of grade on takes`

11.5 Using Triggers Wisely

11.5.1 Good Uses

- Logging changes to a history/audit table
- Recording metadata an application can't see (who ran it, when, from where)
- Simple, self-contained validation
- Keeping denormalized or derived values in sync

11.5.2 Danger Signs

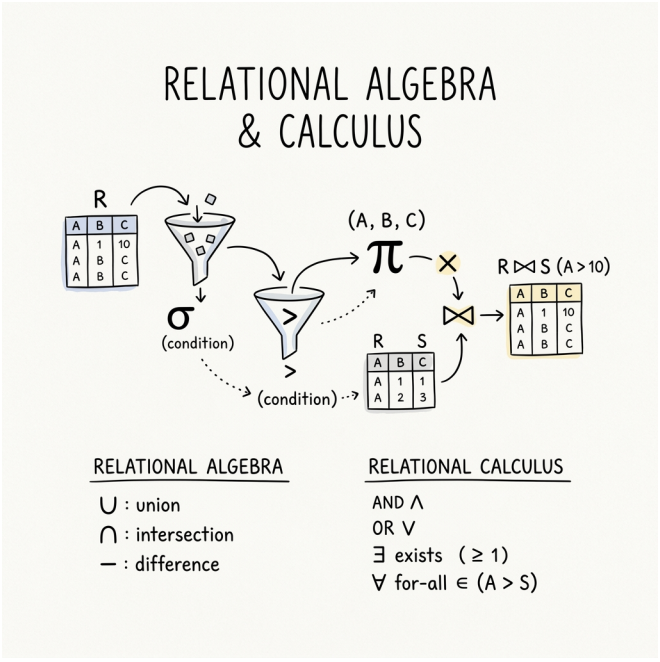
Risk	Why it hurts
Too many triggers	Hard to trace *why* something changed
Complex trigger logic	Business logic buried where developers won't look
Cross-server triggers	Network calls hidden inside what looks like a simple `update`
Triggers calling triggers	Cascading side effects, hard to predict
Recursive triggers enabled	Can spiral — off by default for a reason
Views/procedures/functions nested inside triggers	Performance black box
Iteration inside triggers	Row-by-row processing at scale = slow

Rule of thumb: triggers are best for short, simple, self-contained operations — not as a substitute for your application's business logic.

Part IV

Week-04

12 Relational Algebra and Calculus



13 Introduction to Query Languages

Query languages in DBMS are broadly classified into two categories:

1. **Procedural Query Languages**: The user specifies *what* data is needed and *how* to get it. Relational Algebra is a procedural language.
2. **Non-Procedural (Declarative) Query Languages**: The user specifies *what* data is needed without specifying *how* to get it. Relational Calculus is a non-procedural language.

---

14 Relational Algebra (RA)

Relational Algebra is a procedural query language that takes one or two relations as input and produces a new relation as a result. The fundamental operations include:

- σ - **Select**: Filters rows based on a given condition.
- π - **Project**: Selects specific columns from a relation.
- $\neg$ - **Negation (Not)**
- $\cup$ - **Union**: Combines rows of two relations.
- $\cap$ - **Intersection**: Finds common rows between two relations.
- $\times$ - **Cartesian Product**: Combines each row of the first relation with each row of the second.
- $-$ - **Set Difference (Except)**: Finds rows present in the first relation but not in the second.
- $\bowtie$ - **Natural Join**: Combines two relations based on matching values in common attributes.

14.1 Examples

1. Find all the names of students whose age is greater than 25, or who are enrolled in Maths

Schema: - `Students(RollNo, Name, Age, Subject)`

First, we filter the `Students` table using the Selection (σ) operation for the condition `Age > 25 OR Subject = 'Maths'`. Then, we use the Projection (π) operation to only retrieve the `Name` attribute.

$$\pi_{Name}(\sigma_{Age>25 \vee Subject='Maths'}(Students))$$

2. Find the name and sports of the student whose age is less than 25 and awards is greater than 3

Schema: - `Students(RollNo, Name, Age)` - `Activity(RollNo, Sports, Awards)`

Here we need data from two different domains (the `Students` and `Activity` tables). We use a Natural Join ($\bowtie$) to combine them on their common attribute (`RollNo`), filter the result, and project the specific columns.

$$\pi_{Name,Sports}(\sigma_{Age<25 \wedge Awards>3}(Students \bowtie Activity))$$

14.2 Division Operation

The **Division ($r \div s$)** operation is used in queries that involve the word "every" or "all". For example, "Find students who have taken *all* courses offered in the Biology department."

Suppose we have two relations: $r(A, B, C)$ and $s(B, C)$. The result of $r \div s$ will be a relation with the attribute A, containing values of A that match with *all* the tuples present in s.

- Relations *r, s*:

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

r

D	E
a	1
b	1

s

Figure 14.1: Division Operation Example

In the mathematical example from the notes, if we perform $r \div s$:

A	B	C
α	a	γ
γ	a	γ

The output would be the values in column `A` (α and γ) that are associated with all tuples of the divisor.

---

15 Tuple Relational Calculus (TRC)

Tuple Relational Calculus is a **non-procedural** query language. Instead of writing operations to fetch data, we write mathematical predicates that define the resulting tuples.

A basic TRC query is of the form:

$$\{t|P(t)\}$$

Where: - t = The resulting tuple(s) - $P(t)$ = The predicate (a condition that evaluates to true or false for tuple t)

15.1 TRC Examples

1. Find the name of the students whose age is 21

$$\{t|\exists s \in students(t.name = s.name \wedge s.age = 21)\}$$

Intuition: We are looking for a tuple t such that there exists ($\exists$) a tuple s in the `students` relation where the name in our result matches the name in the student record, and the student's age is 21.

2. Find the name of the employees who work in the 'Manufacturing' department

Schema: - `employee(id, name, salary)` - `department(id, d_id, name, building)`

$$\{M|\exists E \in employee \exists D \in department(E.id = D.id \wedge D.name =' Manufacturing' \wedge M.name = E.name)\}$$

Intuition: We want a tuple M such that there exists an employee E and a department D where their IDs match, the department name is 'Manufacturing', and our result tuple M takes the name of that employee E.

---

16 Domain Relational Calculus (DRC)

Domain Relational Calculus is another non-procedural query language, equivalent in expressive power to Tuple Relational Calculus. However, instead of using whole tuples as variables, DRC uses variables that take values from domains (individual attributes/columns).

A basic DRC query is of the form:

$$\{< x_1, x_2, ..., x_n > | P(x_1, x_2, ..., x_n)\}$$

Where: - $< x_1, x_2, ..., x_n >$ represents domain variables (e.g., individual columns like name, age, etc.) - P represents a predicate formula similar to TRC.

16.1 DRC Examples

1. Find the name of the students whose age is 21

Schema: - `student(name, age, marks)`

$$\{< a > | \exists b(< a, b, c > \in students \wedge b = 21)\}$$

Intuition: We want to output a single domain variable a (which represents the name), such that there exists some marks variable c where the tuple $< a, b, c >$ exists in the `students` relation, and the age variable $b = 21$.

2. Find the name of the employees who work in the 'Manufacturing' department

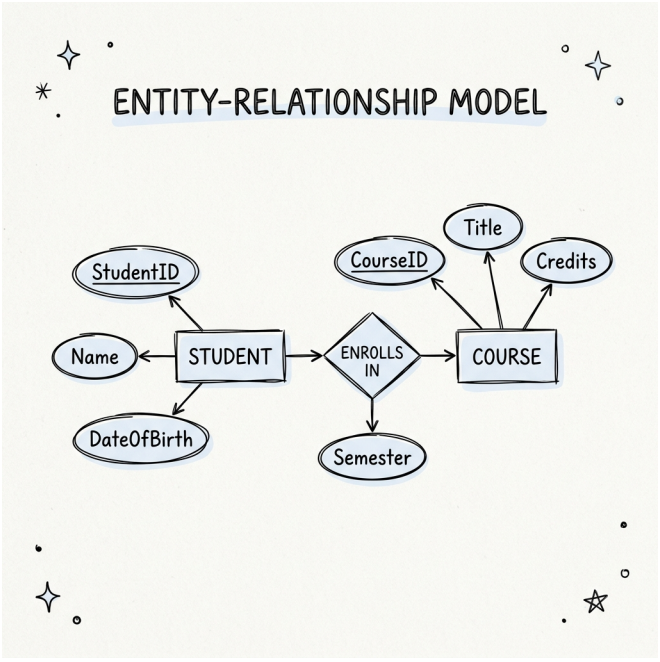
Schema: - `employee(id, name, salary)` - `department(id, d_id, name, building)`

$$\{< b > | \exists a, c(< a, b, c > \in employee) \wedge \exists x, y, z(< a, x, y, z > \in department \wedge y =' Manufacturing')\}$$

(Note: The notes had a slight variable mismatch. I corrected it here to ensure variable a maps to `id`*, b maps to* `name`*, c maps to* `salary`*, and in the department relation, the first variable a corresponds to* `id`*, ensuring the join condition.)*

Intuition: We want to output variable b (employee name). To do this, we assert that the tuple $< a, b, c >$ must exist in `employee`. Additionally, we assert that the tuple $< a, x, y, z >$ must exist in `department` where the variable y (department name) is exactly 'Manufacturing'. The shared variable a (the ID) effectively joins the two relations.

17 Entity-Relationship Model



18 Introduction to Entity Sets

The Entity-Relationship (ER) model is a high-level conceptual data model that helps in designing the schema of a database. It allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships.

- **Entity**: An entity is an object that exists and is distinguishable from other objects. For example, a specific student named "Alice" is an entity.
- **Entity Set**: An entity set is a collection of entities of the same type that share the same properties or attributes. For example, the set of all students in a university forms an entity set.

18.1 Strong Entity Set

- A strong entity set is an entity set that contains sufficient attributes to uniquely identify all its entities.
- A strong entity set always has a **Primary Key**.

18.2 Weak Entity Set

- A weak entity set is an entity set that does not contain sufficient attributes to uniquely identify its entities on its own.
- A primary key does not exist for a weak entity set.
- Instead, it contains a partial key called a **Discriminator**. The discriminator is represented in an ER diagram by underlining the attribute with a **dashed line**.

18.2.1 Characteristics of Weak Entity Sets

- A weak entity set cannot exist independently since it doesn't have a primary key. It depends on a strong entity set.
- It features in the model in relationship with a strong entity set. This dependency is called an **identifying relationship**.
- The primary key of a weak entity set is formed by combining its discriminator with the primary key of the strong entity set it depends on.

- **Primary key of weak entity set = Discriminator + Primary key of Strong entity set**.
- A weak entity set must have **total participation** in its identifying relationship.

---

19 Attributes

An attribute is a property or characteristic associated with an entity set. For example, the `Student` entity set might have attributes like `Name`, `Age`, and `Roll Number`.

19.0.1 Types of Attributes

- **Simple attribute**: An attribute that cannot be divided into smaller sub-parts (e.g., `Age`).
- **Composite attribute**: An attribute that can be divided into smaller sub-parts. For example, a `Name` attribute can be divided into `First Name`, `Middle Name`, and `Last Name`.
- **Single-valued attribute**: An attribute that holds a single value for a specific entity (e.g., `Roll Number`).
- **Multivalued attribute**: An attribute that can hold multiple values for a specific entity. For example, a student might have multiple phone numbers (`{phone_numbers}`).
- **Derived attribute**: An attribute whose value is calculated (derived) from the value of another attribute. For example, `Age()` can be derived from the `Date of Birth` attribute.

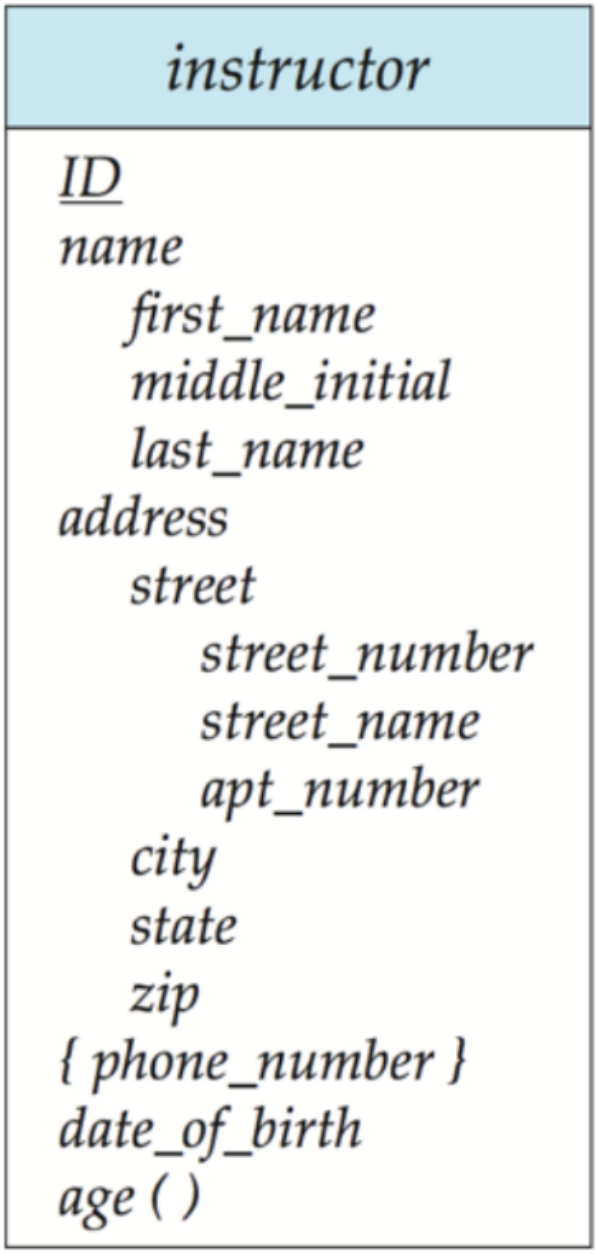


Figure 19.1: Types of Attributes

20 Entity-Relationship (ER) Diagrams

An ER diagram is a visual representation of the ER model. The standard notation involves:

- **Rectangles**: Represent entity sets.
- **Ellipses**: Attributes are listed inside ellipses and connected to the entity set.
- **Solid Underline**: Indicates a primary key attribute.
- **Diamonds**: Represent relationship sets.
- **Double Lines**: Indicate total participation of an entity in a relationship.
- **Double Rectangles**: Represent weak entity sets.
- **Double Diamonds**: Represent identifying relationship sets for weak entity sets.

---

21 Mapping Cardinalities (Relationship Types)

Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.

21.1 One-to-One (1:1) Relationship

- Each entity in A is associated with at most one entity in B.
- Each entity in B is associated with at most one entity in A.
- *Example*: Each instructor has at most one student, and each student has at most one instructor.



Figure 21.1: One to One

21.2 One-to-Many (1:N) Relationship

- Each entity in A can be associated with one or many entities in B.
- Each entity in B is associated with at most one entity in A.
- *Example*: An instructor can advise multiple students, but each student has only one instructor.

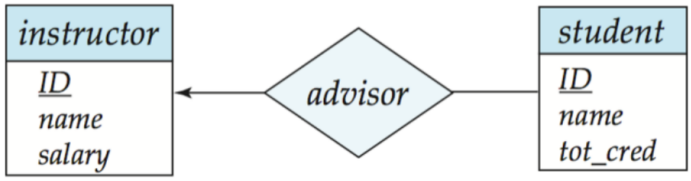


Figure 21.2: One to Many

21.3 Many-to-One (N:1) Relationship

- Each entity in A is associated with at most one entity in B.
- Each entity in B can be associated with one or many entities in A.
- *Example*: Many students can be advised by the same instructor, but each student has at most one instructor.

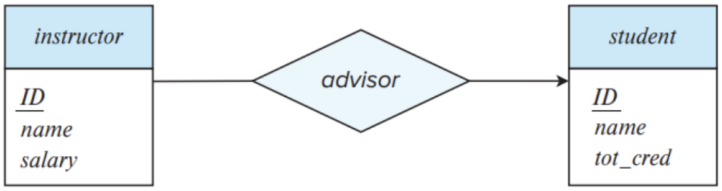


Figure 21.3: Many to One

21.4 Many-to-Many (M:N) Relationship

- Each entity in A can be associated with one or many entities in B.
- Each entity in B can be associated with one or many entities in A.
- *Example*: An instructor can advise multiple students, and a student can have multiple instructors.

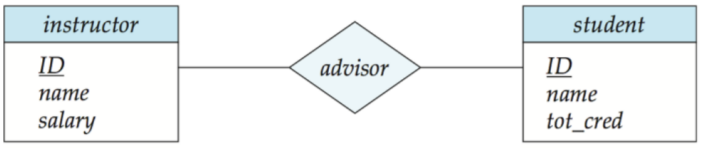


Figure 21.4: Many to Many

---

22 Total and Partial Participation

Participation constraints dictate whether the existence of an entity depends on its being related to another entity via the relationship type.

- **Total Participation**: Every entity in the entity set must participate in at least one relationship in the relationship set. Indicated by a double line.
- **Partial Participation**: Some entities may not participate in any relationship in the relationship set. Indicated by a single line.

Example: If every student *must* have an instructor, the `Student` entity set has total participation. If an instructor *may or may not* have a student, the `Instructor` entity set has partial participation.

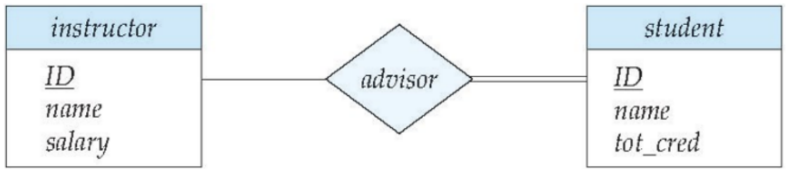


Figure 22.1: Total and Partial Participation

---

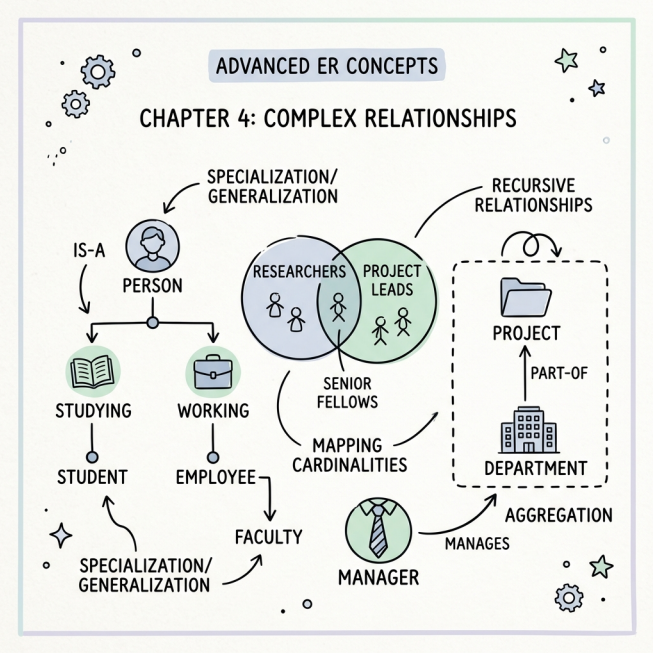
23 Expressing Weak Entity Sets in ER Diagrams

As discussed earlier, a weak entity set does not have a primary key and depends on a strong entity set. In an ER diagram, we denote the weak entity set with a double rectangle, its identifying relationship with a double diamond, and its discriminator attribute with a dashed underline.



Figure 23.1: Weak Entity Set Notation

24 Advanced ER Concepts



As databases grow in complexity, the standard Entity-Relationship model may not be sufficient to capture intricate real-world constraints and relationships. To address this, extended ER features are used.

25 Ternary Relationships

A **Ternary Relationship** is a relationship that exists among three entity sets simultaneously. Instead of linking two entity sets (binary), a single diamond connects three rectangles.

Example: Consider a scenario where `Instructors` assign `Projects` to `Students`. This cannot be accurately captured by breaking it into binary relationships because a specific student is assigned a specific project by a specific instructor. Thus, the relationship involves all three entities at once.

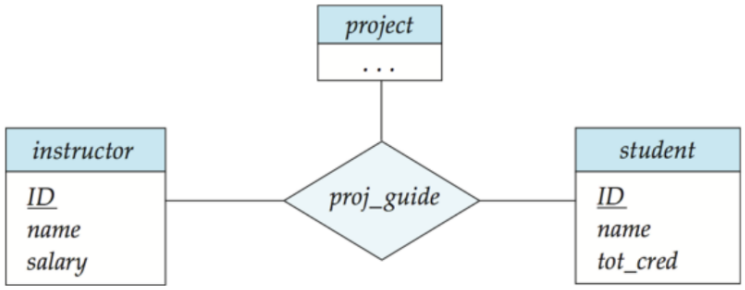


Figure 25.1: Ternary Relationship

26 Aggregation

Aggregation is an abstraction through which relationships are treated as higher-level entities. We use aggregation when we need to establish a relationship *with* another relationship.

Why use Aggregation over Ternary Relationships? Sometimes, a ternary relationship doesn't clearly convey the natural hierarchy of the data. If a relationship inherently acts like an entity that participates in another relationship, aggregation is a better design choice.

Example: Suppose an `Instructor` and a `Student` have an `Advises` relationship. Now, suppose a `Project` is assigned to that specific `(Instructor, Student)` pair. Instead of making a ternary relationship, we draw a box around the `Advises` relationship (aggregating it into a higher-level entity) and connect it to the `Project` entity via an `Assigned` relationship.



Figure: Example of Aggregation

- **Teacher**{*teacher_num*, *teacher_name* }
- **Student**{*student_num*, *student_name* }
- **Project**{*project_num*, *project_name* }
- **Examiner**{*examiner_num*, *examiner_name* }
- **proj_guide**{*teacher_num*, *student_num*, *project_num*, *since* }
- **evaluate**{*teacher_num*, *student_num*, *project_num*, *examiner_num*, *date_of_evaluation* }

Figure 26.1: Aggregation Example

27 Extended ER Features: Specialization and Generalization

Specialization is the process of defining a set of subclasses from an entity set (top-down approach). Generalization is the reverse process of combining subclasses into a higher-level superclass (bottom-up approach).

When dealing with class hierarchies, we must define constraints on how entities can participate in the subclasses. These constraints are categorized by **disjointness** and **completeness**.

27.1 Disjointness Constraints

- **Disjoint**: An entity can belong to *at most one* lower-level entity set (subclass).
- **Overlapping**: An entity can belong to *more than one* lower-level entity set simultaneously.

27.2 Completeness Constraints

- **Complete (Total)**: Every entity in the higher-level entity set (superclass) *must* belong to at least one lower-level entity set.
- **Partial**: Some entities in the higher-level entity set *might not* belong to any lower-level entity set.

Let's look at the combinations of these constraints:

27.2.1 1. Overlapping and Partial

- **Overlapping**: An entity can belong to multiple subclasses.
- **Partial**: An entity does not have to belong to any subclass.

Example: A `Person` can be a `Student`, a `Faculty` member, or **both** (Overlapping). Also, there may be persons (like staff or guests) who are **neither** students nor faculty (Partial).

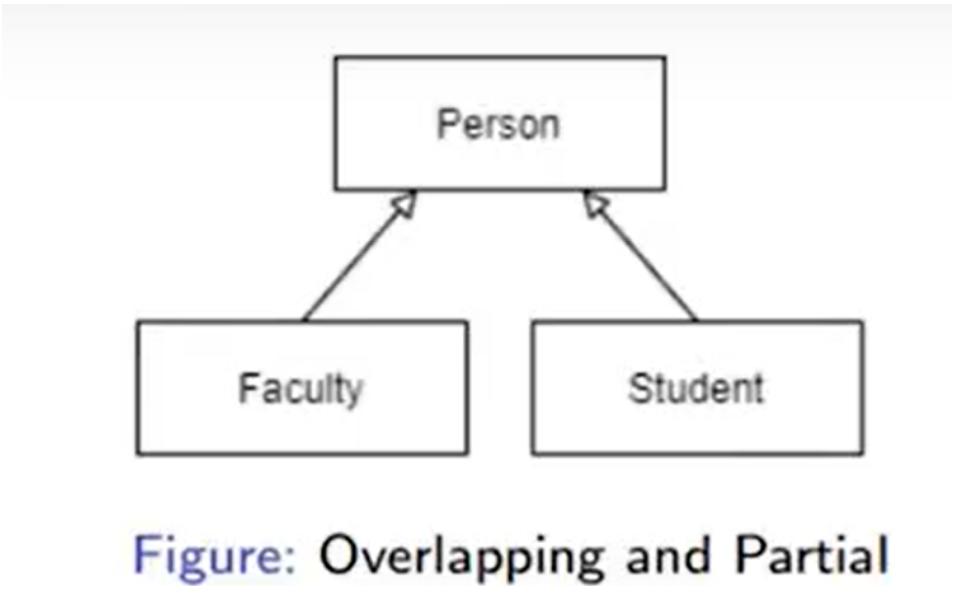


Figure 27.1: Overlapping and Partial

27.2.2 2. Disjoint and Partial

- **Disjoint**: An entity can belong to at most one subclass.
- **Partial**: An entity does not have to belong to any subclass.

Example: A `Student` can be either an `UG_Student` (Undergraduate) or a `PG_Student` (Postgraduate), but they **cannot be both** at the same time (Disjoint). Furthermore, there might be some students (e.g., diploma or visiting students) who are **neither** UG nor PG students (Partial).

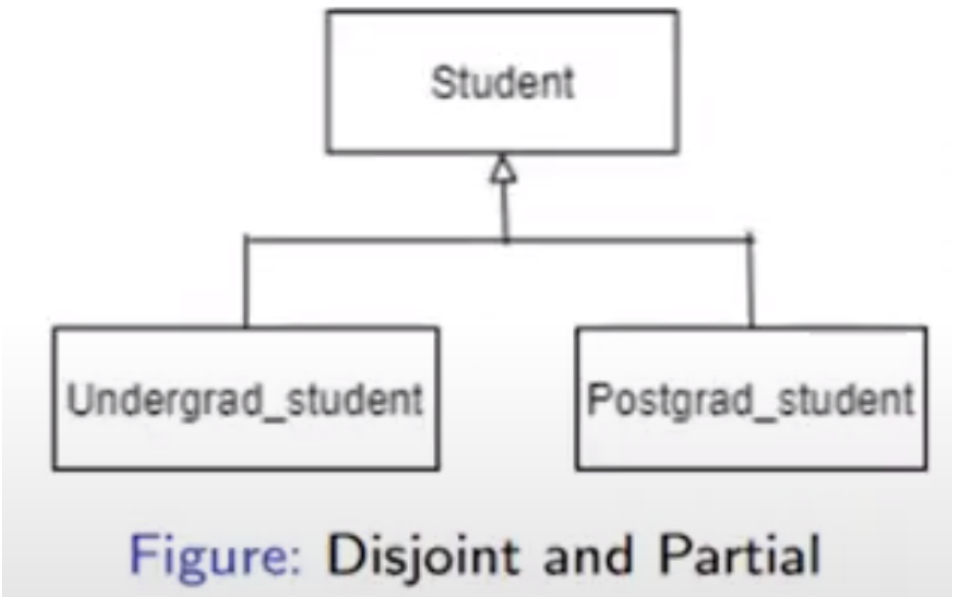


Figure 27.2: Disjoint and Partial

27.2.3 3. Disjoint and Complete

- **Disjoint**: An entity can belong to at most one subclass.
- **Complete**: Every entity must belong to a subclass.

Example: Every physical `Part` in a factory **must** be classified as either a `Purchased_Part` or a `Manufactured_Part` (Complete). A part **cannot** be both purchased and manufactured simultaneously (Disjoint).

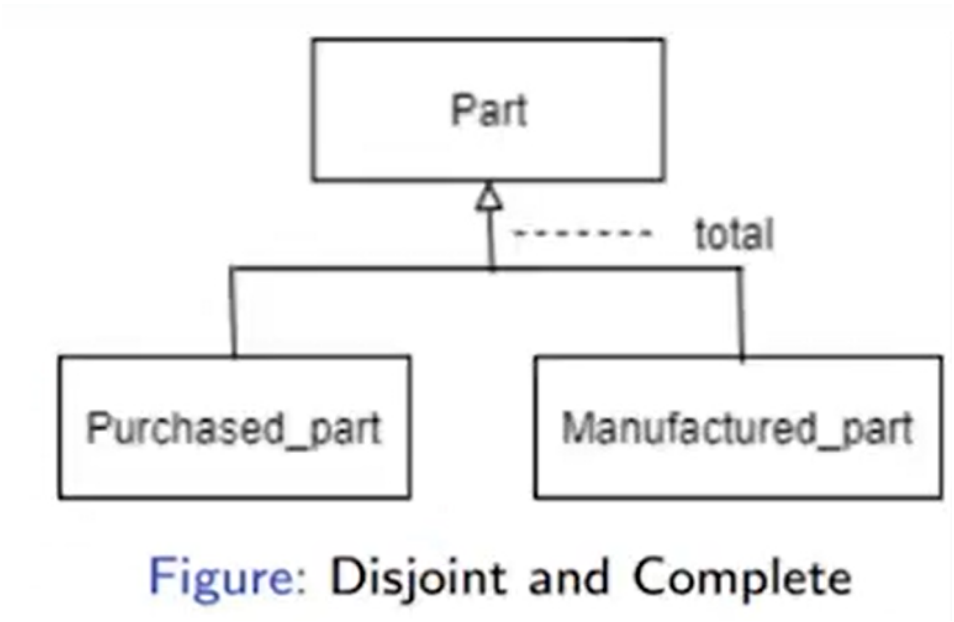


Figure 27.3: Disjoint and Complete

(Note: The fourth combination, Overlapping and Complete, implies an entity must belong to at least one subclass and can belong to multiple. It is less common but conceptually possible.)

---

28 Summary

The concepts covered in the Entity-Relationship model can be summarized below. ER diagrams provide a standard language to communicate database schemas between developers, designers, and stakeholders.

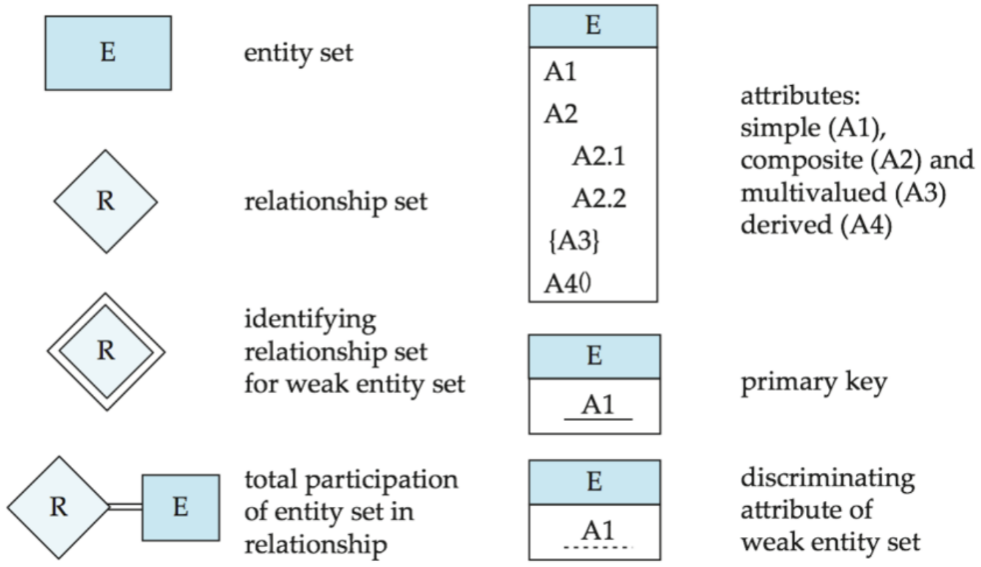


Figure 28.1: ER Diagram Summary

Part V

Week-05

29 Relational Design: Redundancy, Anomalies & Functional Dependencies

29.1 Redundancy and Anomalies

A relational schema should reflect real-world structure, avoid storing the same fact twice, and stay easy to update correctly. When a schema fails at this, two related problems show up: **redundancy** and **anomalies**.

29.1.1 A Bad Schema

`instructor_with_department(ID, name, salary, dept_name, building, budget)`

- `ID`, `name`, `salary` — belong purely to the instructor, no repetition
- `dept_name`, `building`, `budget` — belong to the *department*

Since a department has many instructors, `building` and `budget` get copied once per instructor in that department. That repetition is **redundancy** — the same data item stored in multiple places.

Here's what that looks like with actual rows:

ID	name	salary	dept_name	building	budget
101	Srinivasan	65000	Comp. Sci.	Taylor Hall	100000
102	Wu	90000	Finance	Painter Hall	120000
103	Gold	87000	Comp. Sci.	Taylor Hall	100000
104	Katz	75000	Comp. Sci.	Taylor Hall	100000

Notice `Taylor Hall` and `100000` repeat for every Comp. Sci. instructor — that's the redundancy in action.

29.1.2 Anomalies

Redundancy causes **anomalies** — inconsistencies triggered by insert, delete, or update operations. Using the same table:

Anomaly	What Goes Wrong	On This Table
Insertion	Can't add a record without unrelated data	To add a brand-new "Physics" department with no instructors yet, we can't — there's no row to put it in until a physics instructor is hired
Deletion	Deleting one record erases unrelated info	If Wu (row 102) resigns and we delete that row, we lose the fact that Finance's building is Painter Hall and its budget is 120000 — nowhere else stores that
Update	One fact change needs many row updates	If Comp. Sci.'s budget rises to 110000, we must update rows 101, 103, *and* 104 — miss one, and the table now claims two different budgets for the same department

29.1.3 Why This Happens — and the Fix

`dept_name` uniquely decides `building` and `budget` (a department can't have two budgets). This dependency is exactly what forces `building`/`budget` to repeat across instructor rows:

Dependency → Redundancy → Anomaly

The fix is to **decompose** the table so each fact is stored exactly once — e.g., split `instructor_with_department` into `instructor(ID, name, salary, dept_name)` and `department(dept_name, building, budget)`. (Covered in detail under Lossless Join Decomposition.)

Removing redundancy this way is what makes a schema easier to maintain: updates touch a single row, deletions don't silently destroy unrelated facts, and inserts don't need unrelated data just to go through. This is the core motivation behind everything else in normalization theory.

29.2 Functional Dependencies

29.2.1 Definition

A **functional dependency (FD)** says: knowing the value of one set of attributes is enough to pin down another.

For relation schema R, with $\alpha \subseteq R$ and $\beta \subseteq R$:

$$\alpha \to \beta$$

holds if, for any legal instance of R, two tuples that agree on α must also agree on β:

$$t_1[\alpha] = t_2[\alpha] \implies t_1[\beta] = t_2[\beta]$$

Example: in `instructor_with_department`, we expect:

```
dept_name → building
dept_name → budget
```

since a department has exactly one building and one budget. We would *not* expect `dept_name → salary` — a department tells you nothing about one specific instructor's salary.

> **Note:** An FD only "holds" if it's true for *every* legal instance of the schema, not just today's data. If the current rows happen to satisfy `name → ID` by coincidence (no two instructors currently share a name), that doesn't mean the FD genuinely holds.

29.2.2 Keys, Redefined Using FDs

- K is a **superkey** for R if and only if $K \to R$ (K determines every attribute in R)
- K is a **candidate key** for R if and only if $K \to R$, **and** no proper subset $\alpha \subset K$ also satisfies $\alpha \to R$ (K is minimal)

29.2.3 Trivial FDs

An FD $\alpha \to \beta$ is **trivial** if $\beta \subseteq \alpha$ — it's automatically true for any relation and tells us nothing new.

```
ID, name → ID     -- trivial (ID is already on the left)
name → name       -- trivial
```

30 Armstrong's Axioms & Closure of Attribute Sets

30.1 Armstrong's Axioms

Given a set of FDs F, these three rules let us mechanically derive new FDs that logically follow from it:

Axiom	Rule	Intuition
Reflexivity	if $\beta \subseteq \alpha$, then $\alpha \to \beta$	A set of attributes always determines its own subsets
Augmentation	if $\alpha \to \beta$, then $\gamma\alpha \to \gamma\beta$	Adding the same extra attributes to both sides keeps an FD valid
Transitivity	if $\alpha \to \beta$ and $\beta \to \gamma$, then $\alpha \to \gamma$	If A determines B, and B determines C, then A determines C

Applying these repeatedly to F gives every FD **logically implied** by F — this full set is the closure F^+, and $F \subseteq F^+$.

30.1.1 Worked Example

For $R = (A, B, C, G, H, I)$ with $F = \{A \to B,\ A \to C,\ CG \to H,\ CG \to I,\ B \to H\}$:

$$A \to H \quad \text{(by transitivity, since } A \to B \text{ and } B \to H\text{)}$$

This wasn't stated directly in F, but follows logically — and is therefore part of F^+.

> **Note:** Computing all of F^+ this way (repeatedly applying axioms until nothing new appears) works, but is expensive. **Attribute closure**, covered next, is the practical shortcut.

30.1.2 Derived Rules

These follow from the three basic axioms — not new rules, just useful shortcuts:

Rule	Statement
Union	if $\alpha \to \beta$ and $\alpha \to \gamma$, then $\alpha \to \beta\gamma$
Decomposition	if $\alpha \to \beta\gamma$, then $\alpha \to \beta$ and $\alpha \to \gamma$
Pseudotransitivity	if $\alpha \to \beta$ and $\gamma\beta \to \delta$, then $\alpha\gamma \to \delta$

30.2 Closure of Attribute Sets

Instead of computing the entire F^+, we usually just want to know: what does *this* set of attributes determine? That's the **closure of an attribute set**, written α^+ — all attributes functionally determined by α under F.

30.2.1 How to Compute It

1. Start with `result =  `.
2. Scan through every FD $\beta \to \gamma$ in F.
3. If β is already contained in `result`, add γ to `result`.
4. Repeat steps 2–3 until no new attributes get added.

30.2.2 Worked Example

For $R = (A, B, C, G, H, I)$ with $F = \{A \to B,\ A \to C,\ CG \to H,\ CG \to I,\ B \to H\}$, compute $(AG)^+$:

Step	Result	Why
1	`AG`	starting point
2	`ABCG`	$A \to B$ and $A \to C$ apply
3	`ABCGH`	$CG \subseteq$ result, so $CG \to H$ applies
4	`ABCGHI`	$CG \to I$ applies

$(AG)^+ = ABCGHI$ — every attribute in R.

30.2.3 Superkeys and Candidate Keys, via Closure

- K is a **superkey** of R if $K^+ \supseteq R$ (contains every attribute)
- K is a **candidate key** of R if $K^+ \supseteq R$ **and** no proper subset of K also satisfies this (minimal)

Applying this to AG: $(AG)^+ = R$, so AG is a superkey. Checking subsets — $(A)^+ = ABC$ and $(G)^+ = G$ — neither reaches all of R. So AG is minimal, making it a **candidate key**.

30.2.4 Prime and Non-Prime Attributes

- **Prime attribute**: belongs to *some* candidate key
- **Non-prime attribute**: belongs to *no* candidate key

Here's a data instance for $R = (A, B, C, G, H, I)$ consistent with $F = \{A \to B,\ A \to C,\ CG \to H,\ CG \to I,\ B \to H\}$, compute $(AG)^+$:

A	B	C	G	H	I
a1	b1	c1	g1	h1	i1
a1	b1	c1	g2	h1	i2
a2	b2	c2	g1	h2	i3
a2	b2	c2	g2	h2	i4

Only the pair (A, G) is unique across every row — no other single column or combination pins down a row on its own (`A` alone repeats across rows 1–2 and 3–4; `G` alone repeats across rows 1&3, 2&4). That matches what we found algebraically: AG is the candidate key.

So: `A` and `G` are **prime** attributes; `B`, `C`, `H`, `I` are **non-prime** — each of them is simply *determined by* the key, not part of it.

30.2.5 Formula: Maximum Number of Superkeys

Given a relation with N total attributes, the number of possible superkeys depends directly on the number of candidate keys and the size of those keys.

30.2.6 Case 1: Only 1 Candidate Key (Single Attribute)

If the relation has only one candidate key, and that key consists of a single attribute (e.g., Key $= \{A_1\}$):

$$\text{Total Superkeys} = 2^{N-1}$$

30.2.7 Case 2: Only 1 Candidate Key (Composite Key)

If the relation has only one candidate key, and that key is a composite key formed by k attributes:

$$\text{Total Superkeys} = 2^{N-k}$$

30.2.8 Case 3: Every Single Attribute is a Candidate Key

If the relation has N candidate keys (meaning every single attribute in the relation is its own independent candidate key):

$$\text{Total Superkeys} = 2^N - 1$$

30.2.9 Case 4: Two Distinct Candidate Keys

When a relation contains two distinct candidate keys (which can be composed of one or many attributes), we must use the **Principle of Inclusion-Exclusion** to avoid double-counting the superkeys that contain both keys.

30.2.10 Generic Set Formula

Max Superkeys = (Superkeys with A_1)+(Superkeys with A_2)−(Superkeys with both A_1 and A_2)

30.2.11 Mathematical Representation

Let m be the number of attributes forming the first candidate key, and n be the number of attributes forming the second candidate key. Assuming the two candidate keys share no overlapping attributes:

$$\text{Max Superkeys} = 2^{N-m} + 2^{N-n} - 2^{N-(m+n)}$$

> **Note:** If the two candidate keys share overlapping attributes, the exponent in the subtraction term changes from $(m+n)$ to the number of attributes in their absolute union: $|\text{Key}_1 \cup \text{Key}_2|$.

Example: $R = (A, B, C, G, H, I)$ has $n = 6$ attributes; candidate key AG has $k = 2$. Number of superkeys: $2^{(6-2)} = 2^4 = 16$.

30.2.12 Why Attribute Closure Matters

Use Case	How	In Plain English
Test if α is a superkey	Check $\alpha^+ \supseteq R$	Compute what α determines — if it covers every column in the table, α is enough to identify a row
Test if $\alpha \to \beta$ holds	Check $\beta \subseteq \alpha^+$	Compute what α determines, then see if β shows up in that list
Compute all of F^+	For every $\gamma \subseteq R$, output $\gamma \to S$ for each $S \subseteq \gamma^+$	Try every possible set of columns, work out what each one determines, and write down all those dependencies

31 Extraneous Attributes, Equivalence & Canonical Cover

We already have **attribute closure** (α^+) as our go-to tool for testing superkeys and FDs. Now we use it to ask a sharper question: does every attribute in every FD actually earn its place? This leads to **extraneous attributes** and, from there, the **canonical cover** — the leanest version of an FD set.

31.1 Extraneous Attributes

> **In plain English:** an extraneous attribute is dead weight — remove it, and the FD set means exactly the same thing. It was never pulling any weight to begin with.

An attribute can be dead weight on either side of an FD:

- **Extraneous on the left (α):** attribute $A \in \alpha$ is extraneous in $\alpha \to \beta$ if the *rest* of the left side, without A, still determines β on its own.
- **Extraneous on the right (β):** attribute $A \in \beta$ is extraneous in $\alpha \to \beta$ if the *other* FDs in the set can already derive A anyway — this one FD isn't needed to get it.

31.1.1 How to Test

Left side: Drop A from α. Compute $(\alpha - A)^+$. If β is inside that result, A was extraneous.

Right side: Drop A from β (just for this one FD). Recompute α^+ using this adjusted set. If A still shows up in the result, it means the other FDs already covered it — so A was extraneous.

31.1.2 Example — Extraneous on the Left

$F = \{A \to C,\ AB \to C\}$

Is B extraneous in $AB \to C$? Drop it and check A^+. Since $A^+ = AC$ already contains C, yes — A alone was always enough. B was dead weight.

31.1.3 Example — Extraneous on the Right

$F = \{A \to C,\ AB \to CD\}$

Is C extraneous in $AB \to CD$? Drop it, leaving $\{A \to C,\ AB \to D\}$. Compute AB^+ under this reduced set: it comes out to $ABCD$ — which still contains C. So C was always derivable anyway, and was extraneous in the original FD.

31.2 Equivalence & Canonical Cover

31.2.1 Equivalence of FD Sets

> **In plain English:** two FD sets are equivalent if they're just two different ways of writing down the same set of rules.

F and G are **equivalent** if $F^+ = G^+$. This needs both directions to hold:

- F **covers** G: everything in G can be derived from F
- G **covers** F: everything in F can be derived from G

Only if both hold are F and G truly equivalent.

31.2.2 Canonical Cover

> **In plain English:** the canonical cover is the FD set with all the fat trimmed off — no whole dependency that another one already implies, and no attribute inside a dependency that isn't actually needed.

F_c is a canonical cover of F if:

- $F_c^+ = F^+$ (says exactly the same thing as F)
- No FD in F_c has an extraneous attribute
- Every left-hand side in F_c appears only once (no two FDs share a left side — those would just merge)

31.2.3 How to Compute It

1. Combine any FDs that share the same left-hand side into one, using the union rule.
2. Look at each remaining FD and test both sides for extraneous attributes; remove any that are found.
3. Repeat steps 1–2 until nothing changes.

> Removing an extraneous attribute can create a *new* opportunity to merge left-hand sides that didn't match before — so keep looping until the set is stable.

31.2.4 Worked Example

$R = (A, B, C)$, $F = \{A \to BC,\ B \to C,\ A \to B,\ AB \to C\}$

1. **Merge:** $A \to BC$ and $A \to B$ share a left side $\to$ combine into $A \to BC$. Set is now $\{A \to BC,\ B \to C,\ AB \to C\}$.
2. **Left-side check:** is A extraneous in $AB \to C$? Drop it — $B \to C$ is already in the set, so yes. Set is now $\{A \to BC,\ B \to C\}$.
3. **Right-side check:** is C extraneous in $A \to BC$? Drop it and check — $A \to B$ and $B \to C$ together still give $A \to C$ by transitivity. So yes, C was extraneous.

Canonical cover: $\{A \to B,\ B \to C\}$ — the smallest set that says exactly what the original four FDs said.

32 Lossless Join Decomposition & Dependency Preservation

32.1 Lossless Join Decomposition

> **In plain English:** if you split a table in two, you should be able to join the pieces back together and get *exactly* the original table back — not extra, made-up rows, and not missing ones.

For R decomposed into R_1, R_2, this must hold for every legal instance:

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$$

Sufficient condition: the decomposition is lossless if at least one of these is true:

$$R_1 \cap R_2 \to R_1 \quad \text{or} \quad R_1 \cap R_2 \to R_2$$

In other words: the attribute(s) shared between the two pieces must be enough, on their own, to act as a key for at least one of the two pieces.

> This condition is sufficient whenever all our constraints are FDs. If other kinds of constraints are in play, it isn't guaranteed to be necessary.

Full checklist for a lossless decomposition:

- $R_1 \cup R_2 = R$ — nothing gets left out
- $R_1 \cap R_2 \neq \phi$ — the two pieces actually share something in common
- $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$ — that shared part is a key on at least one side

32.1.1 Worked Example: Supplier-Parts

`Supplier_Parts(S#, Sname, City, P#, Qty)`, with `S# → Sname, City` and `(S#, P#) → Qty`.

Here's a sample instance:

S#	Sname	City	P#	Qty
S1	Smith	London	P1	300
S1	Smith	London	P2	200
S2	Jones	Paris	P1	400
S2	Jones	Paris	P2	300

A bad split: `Supplier(S#, Sname, City, Qty)` and `Parts(P#, Qty)`

The shared attribute here is `Qty` — and `Qty` isn't a key of either piece. Projecting our sample data onto these two tables and joining them back on `Qty` gives:

S#	Sname	City	P#	Qty	
S1	Smith	London	P1	300	original
S1	Smith	London	P2	300	spurious
S2	Jones	Paris	P1	300	spurious
S2	Jones	Paris	P2	300	original
S1	Smith	London	P2	200	original
S2	Jones	Paris	P1	400	original

Both suppliers happened to have a `Qty = 300` row, so the join matches *every* `Qty = 300` row in `Supplier` with *every* `Qty = 300` row in `Parts` — creating two fabricated combinations that never existed in the original data. This decomposition is lossy, and it also fails to preserve `(S#, P#) → Qty`.

A good split: `Supplier(S#, Sname, City)` and `Parts(S#, P#, Qty)`

The shared attribute is `S#`, which *is* a key of `Supplier`. Joining these two projections back together on `S#` reconstructs exactly the original four rows — no more, no less. Lossless, and every dependency is preserved too.

> The lesson: it's not *whether* you split a table, it's *where* you split it. The same table can be decomposed well or badly.

32.2 Dependency Preservation

> **In plain English:** after splitting a table, can you still check all your original rules just by looking at each piece separately — or do you need to join tables back together first to catch a rule violation? If you can check everything locally, the decomposition "preserves dependencies."

Let F_i be the set of FDs from F^+ that only involve attributes inside R_i. A decomposition into $R_1, ..., R_n$ **preserves dependencies** if:

$$(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$$

If this fails, catching a violation of some original FD needs an expensive join across tables.

32.2.1 A Quick Contrast

$R = (A, B, C)$, $F = \{A \to B,\ B \to C\}$

Decomposition	Dependency Preserving?
$R_1 = (A, B),\ R_2 = (B, C)$	Yes — both `A→B` and `B→C` sit fully inside one piece each
$R_1 = (A, B),\ R_2 = (A, C)$	No — `B→C` can't be checked without joining, since `B` and `C` now live in different tables

Same relation, same FDs — the *choice* of split decides the outcome.

32.2.2 How to Test It

1. Start with `result` set to just the left-hand-side attributes (α) of the FD you're checking.
2. Go through each piece R_i of the decomposition. Take the attributes `result` shares with R_i, compute their closure under the *full* F, then keep only the attributes that also belong to R_i. Add anything new to `result`.
3. Repeat step 2, cycling through every R_i, until `result` stops growing.
4. If the right-hand-side attributes (β) are now inside `result`, the FD is preserved without needing a join.

Run this check on every FD in F to confirm the whole decomposition preserves dependencies.

32.2.3 Worked Example

$R(A, B, C, D)$, $F = \{A \to B,\ B \to C,\ C \to D,\ D \to A\}$

Decomposition: $R_1(A, B),\ R_2(B, C),\ R_3(C, D)$

`A→B`, `B→C`, `C→D` each sit fully inside one piece, so they're automatically preserved. The one worth checking is `D→A`:

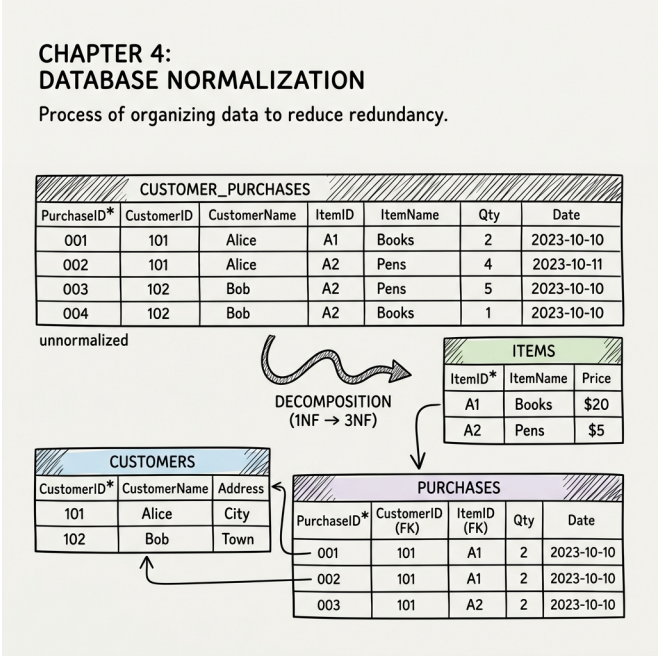
Round	Piece Checked	Closure Computed	Result So Far
Start	—	—	`{D}`
1	$R_1 = (A, B)$	`result  R1 = {}` → nothing to add	`{D}`
1	$R_2 = (B, C)$	`result  R2 = {}` → nothing to add	`{D}`
1	$R_3 = (C, D)$	`(D)+ = ABCD`, keep only `C,D`	`{C,D}`
2	$R_2 = (B, C)$	`(C)+ = ABCD`, keep only `B,C`	`{B,C,D}`
2	$R_1 = (A, B)$	`(B)+ = ABCD`, keep only `A,B`	`{A,B,C,D}`

`result` now contains `A` — so `D → A` is preserved. Every FD in F checks out, so this decomposition is dependency preserving.

Part VI

Week-06

33 Introduction to Normalization



34 What is Normalization?

Normalization (or Schema Refinement) is a systematic technique of organizing data in a database. It involves decomposing a large, poorly-structured table into smaller, well-structured tables.

Normalization serves two primary purposes:

1. **Eliminating Redundant (useless) Data**: Ensuring that the same piece of information isn't stored in multiple places.
2. **Ensuring Data Dependencies**: Ensuring that data is logically stored and related correctly.

---

35 Why Normalize? The Data Anomalies

If a database is not normalized, it can suffer from three major types of **Anomalies** (errors or inconsistencies) when data is modified.

Imagine a single unnormalized table storing both student details and the courses they are enrolled in: `Student_Course(StudentID, StudentName, CourseID, CourseName)`.

35.0.1 1. Insertion Anomaly

An insertion anomaly occurs when you cannot insert certain data into the database without the presence of other unrelated data.

- **Example**: If a new course is created but no students have enrolled in it yet, you cannot add the course to the `Student_Course` table because you would have to leave the `StudentID` (part of the primary key) blank.

35.0.2 2. Deletion Anomaly

A deletion anomaly occurs when deleting a row inadvertently causes the loss of other unrelated information.

- **Example**: If a student is the only person enrolled in "Advanced Databases", and that student drops out, deleting their record will also completely erase the existence of the "Advanced Databases" course from the system.

35.0.3 3. Update Anomaly

An update anomaly occurs when a piece of data is stored in multiple rows, and updating it requires changing multiple records. If one record is missed, the database becomes inconsistent.

- **Example**: If "Alice" gets married and changes her name, you must update her `StudentName` in every single row for every course she takes. If you miss one, the database will show two different names for the same `StudentID`.

---

36 Key Prerequisites

Before diving into Normal Forms, we must define the mathematical rules that govern how attributes relate to one another.

36.1 Functional Dependencies (FDs)

A Functional Dependency, denoted as $X \rightarrow Y$, means that the attribute(s) X uniquely determines the attribute(s) Y. If you know the value of X, you can look up exactly one corresponding value of Y.

- **Example**: $StudentID \rightarrow StudentName$. If you know the ID, there is only one possible name associated with it. Here, X is called the **Determinant**.

36.2 Database Keys

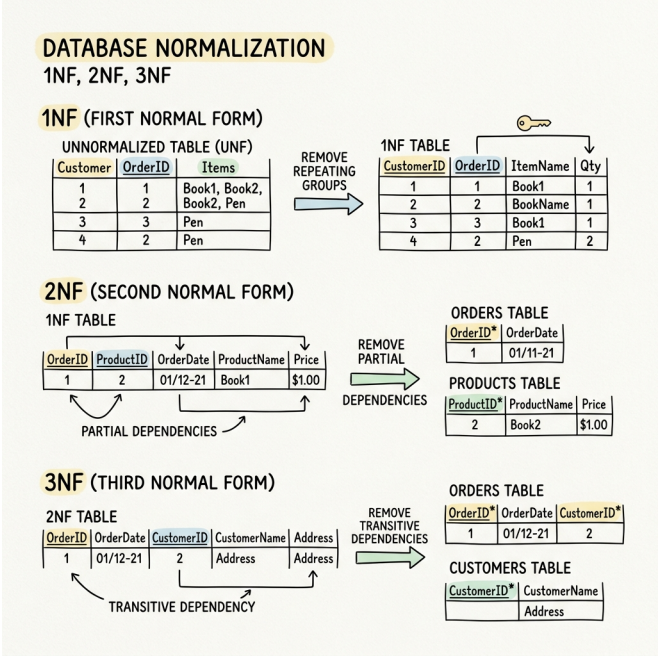
- **Superkey**: A set of one or more attributes that can uniquely identify a row in a table. If $\{StudentID\}$ is a superkey, then $\{StudentID, StudentName\}$ is also a superkey.
- **Candidate Key**: A *minimal* superkey. It is a superkey where no proper subset of its attributes can uniquely identify a row. There can be multiple candidate keys in a table.
- **Primary Key**: The specific candidate key chosen by the database designer to uniquely identify rows.

36.3 Attribute Types

Based on the Candidate Keys, all attributes in a table are divided into two categories:

- **Prime Attribute**: An attribute that is a part of *any* candidate key.
- **Non-Prime Attribute**: An attribute that is *not* a part of any candidate key.

37 Normal Forms (1NF to 3NF)



A **Normal Form** specifies a set of conditions that a relational schema must satisfy in terms of its constraints. As we move from 1NF up to higher normal forms, the rules become stricter, and data redundancy is further reduced.

---

38 1NF (First Normal Form)

A relation is in First Normal Form (1NF) if and only if it satisfies the following rules:

1. **Atomic Values**: All domains should only have atomic (indivisible) values.
2. **No Multivalued Attributes**: Each cell of the table must hold a single value.

38.0.1 Example: Failing 1NF

Consider a table storing student courses:

SID	Sname	Cname
S1	A	C, C++
S2	B	C++, DB
S3	C	DB

- The `Cname` column contains multiple values for S1 ("C", "C++") and S2 ("C++", "DB").
- Because *Cname* is a multivalued attribute, the above table is **not** in 1NF.
- To convert this to 1NF, we would create separate rows for each course a student takes.

---

39 2NF (Second Normal Form)

A relation R is in Second Normal Form (2NF) only if:

1. It is already in **1NF**.
2. It contains **no Partial Dependency**.

39.0.1 Understanding Partial Dependency

A dependency $A \to B$ is a partial dependency only if:

- A is a **proper subset of a Candidate Key**.
- B is a **Non-Prime Attribute** (an attribute not part of any candidate key).

In simple terms: *No non-prime attribute should depend on just a part of a composite candidate key.* If a table has a single-attribute candidate key (and it's the only key), it is automatically in 2NF.

39.1 2NF Solved Example

Consider a relation $R(A, B, C, D, E, F)$ and its Functional Dependencies:

$$FD = \{AB \to CDE, E \to F, BF \to A, C \to B\}$$

Step 1: Find the Candidate Keys Let's find the closures of attribute sets to see which ones determine all attributes in R.

- $(AB)^+ = \{A, B, C, D, E, F\}$
- Since $C \to B$, we can substitute B with C: $(AC)^+ = \{A, C, B, D, E, F\}$
- Since $BF \to A$, if we have B and F, we get A. Thus $(BF)^+ = \{B, F, A, C, D, E\}$

Our **Candidate Keys** are: $\{A, B\}, \{A, C\}, \{B, F\}$.

Step 2: Identify Prime and Non-Prime Attributes

- **Prime Attributes**: A, B, C, F (since they appear in at least one candidate key).
- **Non-Prime Attributes**: D, E.

Step 3: Check for Partial Dependencies We need to check if any non-prime attribute (D or E) depends on a *proper subset* of a candidate key. The proper subsets of our candidate keys are $\{A\}, \{B\}, \{C\}, \{F\}$.

- Do we have $A \to D$ or $A \to E$? No.
- Do we have $B \to D$ or $B \to E$? No.
- Do we have $C \to D$ or $C \to E$? No (C only determines B, which is a prime attribute).
- Do we have $F \to D$ or $F \to E$? No.

Since no non-prime attribute depends on a part of a candidate key, there is **no partial dependency**. **Conclusion**: Relation R **is in 2NF**.

---

40 3NF (Third Normal Form)

A relational schema R is in 3NF if, for every functional dependency $A \to B$ associated with R, **at least one** of the following conditions is true:

1. $B \subseteq A$ (It is a trivial FD).
2. A is a **superkey** of R.
3. B is a **prime attribute** of R.

In simple terms: *3NF prohibits transitive dependencies where a non-prime attribute determines another non-prime attribute.*

40.0.1 Note:

- 3NF guarantees both **dependency preservation** and **lossless join decomposition**.

40.1 3NF Solved Example

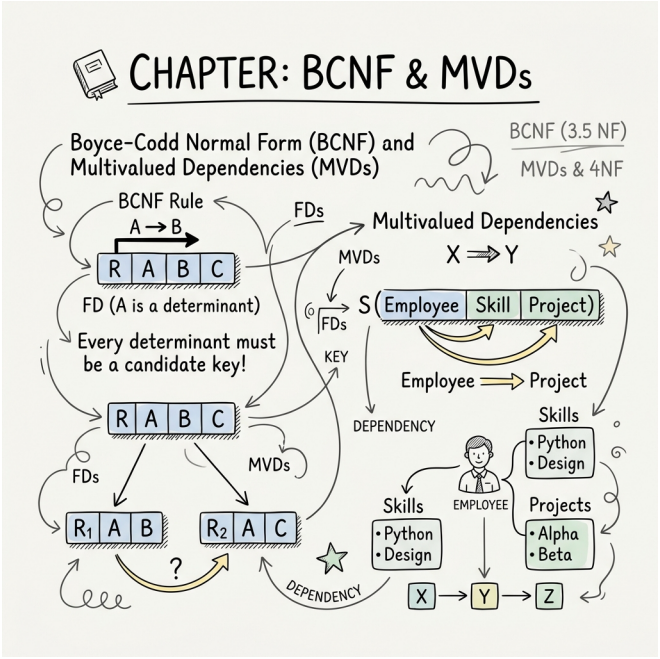
Let's use the same relation $R(A, B, C, D, E, F)$ with $FD = \{AB \to CDE, E \to F, BF \to A, C \to B\}$. We know it is in 2NF. We know the Candidate Keys are $\{A, B\}, \{A, C\}, \{B, F\}$, and Prime Attributes are A, B, C, F.

Let's check each FD against the 3NF rules:

- $AB \to CDE$: AB is a superkey. (Condition 2 holds - Passes).
- $E \to F$: E is *not* a superkey. However, F is a prime attribute! (Condition 3 holds - Passes).
- $BF \to A$: BF is a superkey. (Condition 2 holds - Passes).
- $C \to B$: C is *not* a superkey. However, B is a prime attribute! (Condition 3 holds - Passes).

Conclusion: Since every functional dependency satisfies at least one of the 3NF conditions, the relation R **is in 3NF**.

41 BCNF and Multivalued Dependencies



While 3NF is sufficient for most practical database designs, it still allows certain types of anomalies if there are overlapping candidate keys. Boyce-Codd Normal Form (BCNF) was introduced to handle these edge cases by applying a stricter rule.

---

42 BCNF (Boyce-Codd Normal Form)

A relational schema R is in BCNF if for every non-trivial functional dependency $A \to B$ associated with R, the following condition holds: - A **must be a superkey**.

Simply put: The left side of every functional dependency must uniquely identify the entire row. The "prime attribute" loophole present in 3NF is removed.

42.0.1 Note:

- BCNF decomposition is always **lossless**.
- BCNF decomposition **may or may not be dependency preserving**.

42.1 BCNF Solved Example 1: Success

Consider $R(A, B, C, D)$ and its $FD = \{A \to B, B \to C, C \to D, D \to A\}$

Step 1: Find Candidate Keys

- $A^+ = \{A, B, C, D\}$
- $B^+ = \{B, C, D, A\}$
- $C^+ = \{C, D, A, B\}$
- $D^+ = \{D, A, B, C\}$ Every single attribute is a candidate key (and thus, a superkey).

Step 2: Check BCNF Condition

- For $A \to B$, A is a superkey.
- For $B \to C$, B is a superkey.
- For $C \to D$, C is a superkey.
- For $D \to A$, D is a superkey.

Conclusion: All functional dependencies satisfy the BCNF condition. The relation **is in BCNF**.

42.2 BCNF Solved Example 2: Failure

Let's look back at the example from the 3NF chapter: $R(A, B, C, D, E, F)$ with $FD = \{AB \rightarrow CDE, E \rightarrow F, BF \rightarrow A, C \rightarrow B\}$. We found the candidate keys were $\{A, B\}, \{A, C\}, \{B, F\}$.

Now let's check the FDs against the strict BCNF rule:

- $E \rightarrow F$: Is E a superkey? **No.**
- $C \rightarrow B$: Is C a superkey? **No.**

Conclusion: Because E and C are not superkeys, the relation **fails BCNF**, even though it was successfully in 3NF!

---

43 Multivalued Dependency (MVD)

As we move beyond BCNF, we encounter anomalies caused by independent multivalued facts about an entity. This introduces the concept of Multivalued Dependency, denoted as $\twoheadrightarrow$.

Let R be a relation schema. The multivalued dependency $X \twoheadrightarrow Y$ holds if, for a single value of X, there is a set of values for Y that are completely independent of the other attributes in the table.

For an MVD to exist:

- The total number of attributes in the table should be more than two.
- If there are exactly 3 attributes (X, Y, Z), then Y and Z must be independent of each other, but both depend on X.

43.1 Visualizing MVD

course_name	instructor	book	edition
DBMS	Geeta	DBMS-Beginner	3
DBMS	Arjun	DBMS-Beginner	3
DBMS	Geeta	DBMS-Expert	2
DBMS	Arjun	DBMS-Expert	2
Java	Rahul	Java-Beginner	5
Java	Rahul	Java-Intermediate	3
Java	Rahul	Java-Expert	4
Java	Armaan	Java-Beginner	5
Java	Armaan	Java-Intermediate	3
Java	Armaan	Java-Expert	4

Figure 43.1: MVD Example

In this example, an instructor teaches a course, and a course has recommended books.

- The instructors for a course are completely independent of the books for a course.

- Thus, `course_name` $\twoheadrightarrow$ `instructor`
- And `course_name` $\twoheadrightarrow$ `{book, edition}`

If you try to store this in a single table, you must repeat every instructor for every book, causing massive data redundancy!

43.2 Trivial MVD

An MVD $X \twoheadrightarrow Y$ is considered **trivial** if:

- Y is a subset of X ($Y \subseteq X$)
- Or, $X \cup Y = R$ (the entire table)

---

44 MVD Theory Rules

MVDs follow a set of inference rules, similar to regular functional dependencies:

Name	Rule
Complementation	If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow (R - (X \cup Y))$
Augmentation	If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$
Transitivity	If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$
Replication	If $X \to Y$, then $X \twoheadrightarrow Y$ (but the reverse is not true)
Coalescence	If $X \twoheadrightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \to Z$, and $Z \subseteq Y$, then $X \to Z$

45 4NF (Fourth Normal Form)

To resolve the redundancies caused by Multivalued Dependencies, we use the Fourth Normal Form.

A relation schema R is in 4NF if and only if the following conditions are satisfied:

1. R is already in **BCNF**.
2. It does not contain any **non-trivial multi-valued dependencies**.

If a table violates 4NF due to $X \twoheadrightarrow Y$, we fix it by decomposing the table into two separate tables: one holding (X, Y) and the other holding $(X, \text{everything else})$.

Part VII

Week-07

46 Application Programs and Architecture

46.1 Application Programs

Databases don't exist in isolation, they always live inside an **application** that users interact with. Database-backed applications show up in almost every corner of modern life, for example, travel, communication, education etc. Despite looking completely different, most are built the same way underneath — they all use an RDBMS and they are all split into three layers.

46.2 The Three-Layer Architecture

Think of it like a restaurant : the waiter takes your order (Presentation), the chef decides how to prepare it (Business Logic), and the pantry stores all the ingredients (Data Access). Each role is distinct, and each has its own concerns.

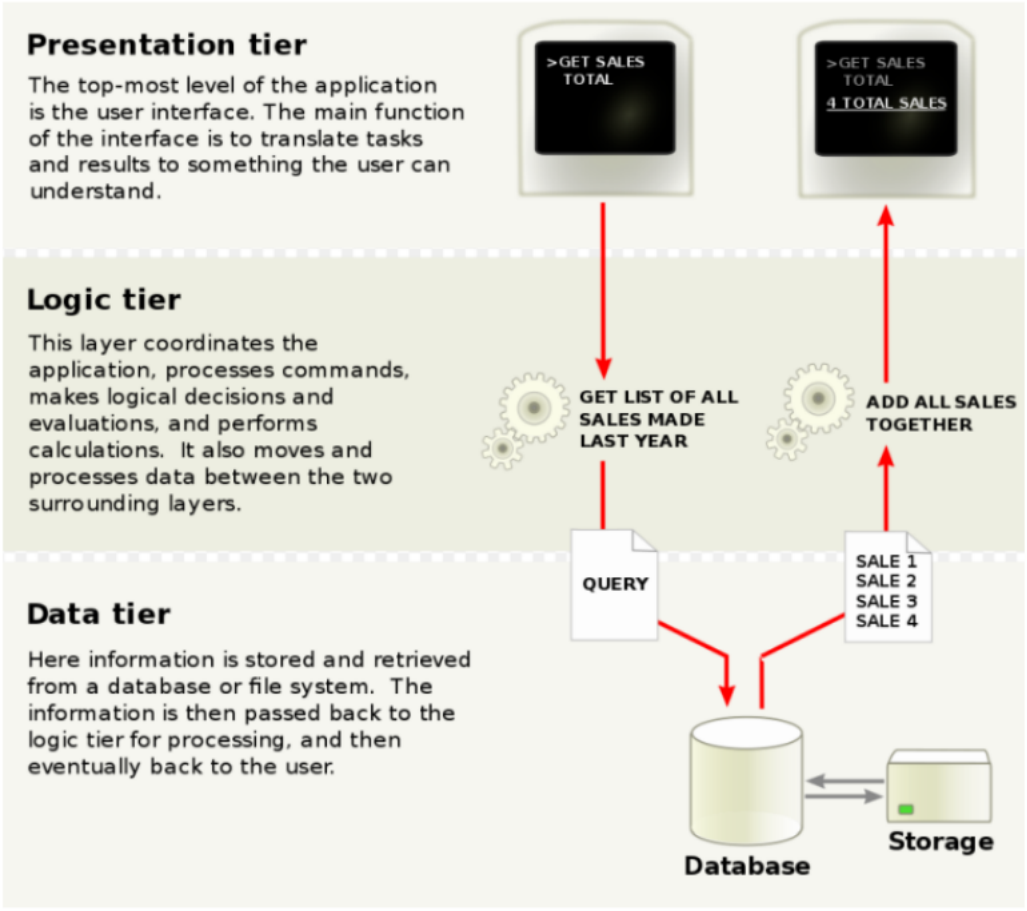


Figure 46.1: Three-Layer Architecture

46.2.1 Presentation Layer (Frontend)

This is everything the user actually sees and touches. It collects input and displays results. Within the presentation layer, the most widely used design pattern is **Model-View-Controller (MVC)**:

Component	Role
Model	Represents business data and rules
View	Displays data to the user (depends on device/format)
Controller	Receives user events, invokes model actions, selects the view to return

46.2.2 Business Logic Layer (Middle Layer / Tier 2)

Implements the core functionalities, workflows, and rules of the application (such as authentication, search logic, pricing, payment gateways, and order management). It acts as the bridge that connects the frontend and backend layers.

> The middle layer hides the database schema from the frontend. The frontend never needs to know which table a piece of data lives in.

46.2.3 Data Access Layer (Backend / Tier 3)

Manages persistent data. It is responsible for handling large volumes of information, ensuring high-speed access, maintaining data security, and interfacing directly with the database system .

46.3 Architecture Classification

Tier	What it looks like
1-Tier	All components (user interface, middleware, and backend database) reside on a single server or platform. This is the simplest and most direct setup
2-Tier	A classic client-server model where the client directly communicates with the database server with no intermediate application logic server.
3-Tier	Presentation, Business Logic, and Data Access are separated into three independent layers. This is the de-facto standard for DBMS design.
n-Tier	Extends 3-tier by distributing components across many servers with load balancers, caches, and microservices

---

46.4 Web Applications and Scripting

46.4.1 The World Wide Web

The Web is a massive collection of **hypertext documents** (HTML pages) spread across servers worldwide, all linked to each other. HTML can contain text, images, hyperlinks, and forms — input fields that let users send data back to a server, which is the bridge between the user and the database.

46.4.2 URLs, URIs, and URNs

Every resource on the web needs an address. A URL like `http://www.acm.org/sigmod` has three parts:

```
http  ://  www.acm.org  /sigmod
 ↑              ↑           ↑
Protocol    Machine name   Document path
```

- **URI** — An umbrella term that covers both locators (URLs) and names (URNs).
- **URL** — Specifies how to retrieve a resource and where it is located, functioning like a person's *street address*.
- **URN** — Serves as a unique identifier for a resource, functioning like a person's *name*. It defines the identity of a resource but does not specify its location

---

46.4.3 HTML and HTTP

- **HTML (Hypertext Markup Language)** : Used to format content, embed links, display tables, and gather user input through interactive forms (including dropdown menus, text boxes, and buttons).
- **HTTP (Hypertext Transfer Protocol)** : The primary communication protocol used between web clients and web servers . The key thing to know: HTTP is **connectionless** — once the server replies, it immediately forgets the conversation ever happened. Every new request is treated as brand new. This keeps server load manageable (imagine maintaining open connections with millions of users at once).

46.4.4 Sessions and Cookies

HTTP being connectionless creates a problem — how does a website remember you're logged in as you move between pages?

The answer is **cookies**. Here's how it works:

1. On your first visit, the server sends a small identifying token (cookie) to your browser
2. Your browser stores it and automatically attaches it to every subsequent request
3. The server reads it, looks it up, and knows exactly who you are

This is how Amazon keeps your cart alive across sessions. Cookies can be **permanent** (survive browser restarts) or **session-only** (deleted when you close the tab).

46.5 Scripting

A **script** is a set of commands embedded in a web page or on the server, executed at runtime. Scripts are what make pages dynamic. Two types:

46.5.1 Client-Side Scripting

Client-side scripts are downloaded directly to the client's computer and run locally inside the web browser.

- **Primary Languages** : JavaScript, VBScript.
- **Purpose** : Handles immediate user interactions, validates form inputs locally, and updates page visuals without triggering a full page reload (e.g., AJAX technology).
- **Security Context** : To prevent malicious scripts from accessing local hard drives, client-side scripts are executed in a heavily restricted **safe-mode sandbox** that prevents direct system calls or unauthorized file operations.

46.5.2 Server-Side Scripting

Server-side scripts are executed entirely on the web server before the final HTML is generated and sent back to the client.

- **Primary Languages** : PHP, JSP, ASP, Python, Perl.

- **Purpose** : Simplifies database integration. Developers embed executable logic or database queries directly within HTML files. The server runs these queries, dynamically inserts the results into the HTML, and returns the finished static webpage to the user.
- **Security Context** : Safe and secure. The client never sees the server's proprietary code or database connection details—only the final, clean HTML output.

46.5.3 Servlets

A **Java Servlet** is a Java program that lives on a web/application server and handles HTTP requests. The flow:

```
User submits form
    → HTTP request hits the server
    → Server loads the Servlet
    → Servlet reads parameters, runs SQL via JDBC
    → Servlet writes HTML response
    → Thread closes
```

Each request gets its own **thread**, so many users can be served simultaneously.

Session handling in Servlets uses cookies automatically:

```
request.getSession(true);                    // create session
session.setAttribute("userid", "alice");     // store data
session.getAttribute("userid");              // retrieve later
```

Servlets run inside application servers like **Apache Tomcat**, GlassFish, or JBoss.

46.5.4 JSP (Java Server Pages)

Writing HTML entirely through `out.println()` in a Servlet is painful. JSP lets you write HTML normally and drop Java logic inside `<% ... %>` tags wherever needed:

```
<body>
  <% if (request.getParameter("name") == null) { %>
    Hello, World!
  <% } else { %>
    Hello, <%= request.getParameter("name") %>!
  <% } %>
</body>
```

The server compiles a JSP page into a Servlet the first time it's requested, so performance is identical — JSP is just much easier to write and maintain.

46.5.5 PHP

PHP works exactly like JSP — embed logic inside HTML, server executes it, client gets plain HTML. It has extensive database libraries (ODBC) and was the dominant language for web development for many years.

```
<body>
  <?php
    if (!isset($_REQUEST['name'])) { echo "Hello World"; }
    else { echo "Hello, " . $_REQUEST['name']; }
  ?>
</body>
```

PHP and JSP are conceptually the same. The choice between them comes down to the tech stack being used.

47 SQL and Native Languages

When an application needs to talk to a database, it uses an **API (Application Programming Interface)** — a standard set of function calls that let you:

- Open a connection to the database server
- Send SQL queries or updates
- Fetch results back into your program's variables

There are two broad approaches to doing this:

Approach	What it means	Examples
Connectionist	SQL is sent as a string through a driver at runtime	ODBC, JDBC
Embedded SQL	SQL is written directly inside the host language code and processed before compilation	Embedded SQL for C, Java (SQLJ)

47.1 ODBC (Open Database Connectivity)

ODBC is a standard API that lets applications talk to *any* database, regardless of which RDBMS is being used. The same Python or C code can connect to MySQL, PostgreSQL, or SQL Server with minimal changes, you just swap the driver.

It was originally developed by Microsoft and works with **C, C++, C#, Visual Basic, and Python**.

A typical ODBC workflow:

- Open a connection to the database
- Send a query or update
- Fetch results
- Close the connection

47.2 JDBC (Java Database Connectivity)

JDBC is the Java equivalent of ODBC. It's the standard way for Java programs to talk to a database.

The workflow is very similar:

```
// 1. Open a connection
Connection con = DriverManager.getConnection(connectionUrl);

// 2. Create a statement object
Statement stmt = con.createStatement();

// 3. Execute a query and get results
ResultSet rs = stmt.executeQuery("SELECT * FROM Production.Product");

// 4. Iterate through results
while (rs.next()) {
    System.out.println(rs.getString("ProductID") + " : " + rs.getString("Name"));
    }
```

- `DriverManager.getConnection()` — opens the connection using a connection URL
- `Statement` — used to send SQL to the database
- `ResultSet` — holds the rows returned by a query; you iterate through it with `rs.next()`

47.3 Connectionist Bridges

Sometimes the driver you need doesn't exist — for example, you want to connect a Java app to a database that only has an ODBC driver. **Bridges** solve this by translating function calls from one standard to another.

Bridge	What it does
JDBC-to-ODBC	JDBC driver internally uses an ODBC driver to reach the database
ODBC-to-JDBC	ODBC driver internally uses a JDBC driver
OLE DB-to-ODBC	Microsoft's OLE DB provider translated to ODBC calls
ADO.NET-to-ODBC	ADO.NET provider (used in .NET apps) translated to ODBC

Think of a bridge as a translator — it lets two systems that speak different languages work together.

47.4 Embedded SQL

With ODBC and JDBC, SQL queries are just strings passed to a function at runtime — the compiler has no idea they're even there. **Embedded SQL** takes a different approach: you write SQL *directly inside* the host language code, and a **pre-processor** converts it into regular function calls before compilation.

Supported in: **C, C++, Java (SQLJ), COBOL, FORTRAN**.

SQL statements are marked with `EXEC SQL` so the pre-processor knows what to handle:

```
EXEC SQL <embedded SQL statement>;
```

47.4.1 Key Concepts

Connecting to the database:

```
EXEC SQL CONNECT TO server USER username USING password;
```

Host variables — variables from your program that you use inside SQL queries. They're prefixed with `:` to distinguish them from SQL column names. Variables used within the Embedded SQL query must be declared within DECLARE section:

```
EXEC SQL BEGIN DECLARE SECTION;
    int credit_amount;
EXEC SQL END DECLARE SECTION;
```

Cursors — since a query can return many rows, you use a *cursor* to step through them one at a time. Think of it as a pointer that moves through the result set.

```
/* 1. Declare the cursor */
EXEC SQL
    DECLARE c CURSOR FOR
    SELECT ID, name FROM student
    WHERE tot_cred > :credit_amount;
```

```
END_EXEC

/* 2. Open it - this actually runs the query */
EXEC SQL OPEN c;

/* 3. Fetch rows one by one */
EXEC SQL FETCH c INTO :si, :sn;

/* 4. Close when done */
EXEC SQL CLOSE c;
```

- `OPEN` — executes the query and stores results in a temporary relation
- `FETCH` — retrieves the next row into your host variables
- `SQLSTATE` is set to `'02000'` when there are no more rows to fetch
- `CLOSE` — releases the temporary result

Updating through a cursor:

You can also modify rows as you fetch them by declaring the cursor `FOR UPDATE`:

```
EXEC SQL DECLARE c CURSOR FOR
    SELECT * FROM instructor
    WHERE dept_name = 'Music'
    FOR UPDATE;

/* After fetching each row: */
UPDATE instructor SET salary = salary + 1000 WHERE CURRENT OF c;
```

47.5 Connecting Python to PostgreSQL

The most popular Python library for working with PostgreSQL is **psycopg2** — it's stable, actively maintained, thread-safe, and works with both Python 2 and 3.

Install it with:

```
pip install psycopg2
```

Every interaction with the database follows the same six steps:

Step	What happens
1. Create connection	`psycopg2.connect(...)` — opens a connection to the DB
2. Create cursor	`conn.cursor()` — creates a cursor to execute queries
3. Execute query	`cur.execute(sql)` — runs your SQL
4. Fetch results	`fetchone()`, `fetchmany()`, or `fetchall()`
5. Commit or rollback	`conn.commit()` to save, `conn.rollback()` to undo
6. Close	`cur.close()` then `conn.close()`

47.6 Key API Methods

Connection:

```
conn = psycopg2.connect(database="mydb", user="myuser",
                        password="mypass", host="127.0.0.1", port="5432")
conn.commit()    # saves changes
conn.rollback()  # undoes changes since last commit
conn.close()     # closes the connection
```

Cursor:

```
cur = conn.cursor()
cur.execute("INSERT INTO employee VALUES (%s, %s)", (id, name))  # parameterized query
cur.rowcount     # number of rows affected by last operation
cur.fetchone()   # returns next row, or None
cur.fetchmany(n) # returns next n rows as a list
cur.fetchall()   # returns all remaining rows as a list
cur.close()
```

> Use `%s` as a placeholder in queries instead of string formatting — this protects against SQL injection.

47.7 CRUD Operations in Python

CREATE table:

```
cur.execute('''CREATE TABLE EMPLOYEE (
    emp_num  INT PRIMARY KEY NOT NULL,
    emp_name VARCHAR(40) NOT NULL,
    department VARCHAR(40) NOT NULL)''')
conn.commit()
```

INSERT:

```
cur.execute("INSERT INTO EMPLOYEE VALUES (%s, %s, %s)", (110, 'Bhaskar', 'HR'))
conn.commit()
```

UPDATE:

```
cur.execute("UPDATE EMPLOYEE SET department = %s WHERE emp_num = %s", ('Finance', 110))
conn.commit()
```

DELETE:

```
cur.execute("DELETE FROM EMPLOYEE WHERE emp_num = %s", (110,))
conn.commit()
```

SELECT:

```
cur.execute("SELECT emp_num, emp_name, department FROM EMPLOYEE")
rows = cur.fetchall()
for row in rows:
    print(row[0], row[1], row[2])
```

---

48 RAD, Application Security, and Mobile Apps

48.1 Rapid Application Development (RAD)

Building rich web application interfaces, especially the kind of interactive, dynamic pages we see today takes a lot of effort. **RAD** is an approach that aims to shorten that development cycle by focusing on fast prototyping, quick feedback, and iterative delivery rather than long planning phases.

The four phases of RAD are:

1. **Business Modeling** — define what the application needs to do
2. **Data Modeling** — design the data structures
3. **Process Modeling** — design the workflows and logic
4. **Testing & Turnover** — test, refine, and deploy

The key idea: prototypes are built quickly, feedback is collected early, and each iteration gets you closer to the final product.

48.2 Application Performance

Popular web applications can have millions of users, with thousands of requests hitting the server every second at peak times. Serving each request by running a fresh database query every single time would be extremely slow and expensive. **Caching** is the main tool used to tackle this.

48.2.1 Where Caching Happens

Location	What gets cached
Server — Connection Pool	JDBC connections are kept open and reused across requests instead of opening a new one each time
Server — Query Results	Results of frequent database queries are stored in memory; the DB is only hit again if the underlying data changes
Server — Generated HTML	If a page looks the same for many users, the rendered HTML is cached and served directly
Client Network — Web Proxy	Pages are cached at the network level and served to nearby users without hitting the server

Connection pooling in particular is a big win — opening a new database connection is expensive, so reusing existing ones across requests saves significant time.

48.3 Application Security

Security is one of the most critical and most commonly overlooked aspects of web application development. Here are the major threats and how to address them.

48.3.1 SQL Injection

This is one of the most common and dangerous attacks. It happens when user input is directly concatenated into a SQL query string without sanitization.

Example of a vulnerable query:

```
"SELECT * FROM instructor WHERE name = '" + name + "'"
```

If a user types `X' or 'Y' = 'Y` as their name, the query becomes:

```
SELECT * FROM instructor WHERE name = 'X' or 'Y' = 'Y'
```

This returns every row in the table. A malicious user could even inject:

```
X'; UPDATE instructor SET salary = salary + 10000; --
```

The fix — always use prepared statements:

```
PreparedStatement ps = conn.prepareStatement(
    "SELECT * FROM instructor WHERE name = ?"
);
ps.setString(1, name);
```

The `?` placeholder ensures the input is treated as data, never as SQL code. Special characters get automatically escaped.

> Note: `conn.prepareStatement("SELECT ... WHERE name = '" + name + "'")` is **not** a prepared statement — it's just a regular string with a misleading name. Always use `?` placeholders.

48.3.2 Password Leakage

- Never store database passwords in plain text inside script files (`.php`, `.jsp`) that live in web-accessible directories.
- Web servers execute script files but may accidentally serve backup files (e.g., `file.jsp~` or `.file.jsp.swp`) as raw text, exposing credentials.
- Restrict database server access to only the IP addresses of your application servers.

48.3.3 Authentication

Passwords alone are increasingly insufficient — they can be guessed, sniffed, reused across sites, or stolen by spyware.

Two-factor authentication (2FA) adds a second layer: - Password + OTP sent via SMS - Password + a hardware token that generates a new pseudo-random number every minute (both the device and the server compute the same sequence independently)

48.3.4 Application-Level Authorization

Standard SQL authorization works at the table or column level — you can say "this user can read the `takes` table", but you can't say "this student can only see *their own* rows". The database has no concept of who the *application user* is.

Workaround — use views:

```
CREATE VIEW studentTakes AS
    SELECT * FROM takes
    WHERE takes.ID = syscontext.user_id();
```

This filters rows based on the currently logged-in user's identity, which the application passes to the database. It works, but creating a view for every such rule gets unwieldy.

48.3.5 Audit Trails

Every sensitive action in an application should be **logged**: who accessed what data, who made what change, and when. Audit trails are used after a breach to detect what happened, repair the damage and trace who was responsible.

Logs are needed at both the **database level** (who ran which queries) and the **application level** (which user performed which action in the UI).

48.4 Challenges in Web Application Development

Building a web application involves more than just writing code. The major challenges are:

- **User Interface & Experience** — making interfaces that are intuitive and work across devices
- **Scalability** — handling growth in users and data without redesigning the system
- **Performance** — keeping response times low under heavy load
- **Framework & Platform Knowledge** — choosing and mastering the right tools
- **Security** — protecting against the threats covered above

---

48.5 Mobile Apps

48.5.1 What is a Mobile App?

A mobile app is application software designed to run on smartphones and tablets. The key difference from desktop development is the set of **constraints** the device imposes:

Constraint	Capability
Small screen / form factor	Touchscreen & gesture-based navigation
Limited memory & CPU	Accelerometer, GPS, camera sensors
Limited battery	Always-on connectivity (WiFi, 4G/5G)
Limited bandwidth	Push notifications

Design must account for both the limitations *and* the unique capabilities.

48.5.2 Architecture of a Mobile App

Mobile apps follow the same **3-tier architecture** as web apps — Presentation, Business Logic, and Data Access. The main difference is in the Data layer, which is often **split**:

- **Local data** — stored on the device (SQLite, cached files) for offline access and speed
- **Remote data** — fetched from a server API when connected

The app also needs to be customized per platform (Android, iOS, Windows), since each has its own OS, UI guidelines, and APIs.

48.5.3 Types of Mobile Apps

Type	How it works	Trade-off
Native	Written in the platform's own language (Java/Kotlin for Android, Swift/Objective-C for iOS)	Best performance and device integration, but separate codebase per platform
Web App	Runs in a browser using HTML/CSS/JS	Fully portable, but limited access to device features
Hybrid	Wraps a web app in a native shell	One codebase for all platforms, but performance trade-offs

Part VIII

Week-08

49 Algorithms & Data Structures

49.1 Algorithms and Complexity Analysis

49.1.1 Basics and Definitions

- **Algorithm vs. Program:** An **algorithm** is a finite sequence of well-defined, computer-implementable instructions to solve a class of problems or to perform a computation. Crucially, an algorithm **must terminate**. A **program**, such as an operating system, is an implementation that may run indefinitely without terminating.

49.1.2 Motivation for Analysis

Why do we analyze algorithms? - To **predict performance** (how fast will this run on a billion rows?). - To **avoid bugs** and **provide guarantees** on execution boundaries. - To understand the theoretical limits of computational problems.

49.1.3 Factors of Analysis

When we analyze an algorithm, we measure the "cost" based on several factors:

1. **Time Complexity:** How long does it take to execute?
2. **Space/Storage Complexity:** How much memory (RAM) does it consume?
3. **Other factors:** Power consumption and network bandwidth (especially relevant in cloud databases).

49.1.4 Methodologies

- **Counting Model:** A simplified theoretical model (Random-Access Machine) where we sum the cost and frequency of primitive operations (like addition, assignment, etc.).
- **Asymptotic Analysis:** We use **Big-Oh (O) notation** to measure the growth of resource requirements as the input size (n) increases toward infinity. We typically focus on the **Worst case** and **Average case** scenarios.

49.1.4.1 Complexity Classes

From fastest (best) to slowest (worst):

1. $O(1)$ - Constant Time
2. $O(\log n)$ - Logarithmic Time
3. $O(n)$ - Linear Time
4. $O(n \log n)$ - Linearithmic Time
5. $O(n^2)$ - Quadratic Time
6. $O(n^3)$ - Cubic Time
7. $O(2^n)$ - Exponential Time

49.2 Data Structures

A **Data Structure** is a specialized format for organizing, processing, retrieving and storing data in memory. Standard operations on any data structure include: **Create, Insert, Delete, Find (Search), Update, and Close.**

49.2.1 Linear Data Structures

1. **Arrays:** Contiguous blocks of memory.
    - *Advantage:* Extremely fast indexed/random access ($O(1)$).
    - *Disadvantage:* Very slow insertion/deletion ($O(n)$) because all subsequent elements must be shifted.
2. **Linked Lists:** Elements are stored in nodes that contain pointers to the next node.
    - *Advantage:* Very fast insertion/deletion ($O(1)$ if you are already at the position).
    - *Disadvantage:* Slow sequential access ($O(n)$) since you must traverse from the head.
3. **Queues:** Follow First-In-First-Out (FIFO) logic (like a line at a grocery store).
4. **Stacks:** Follow Last-In-First-Out (LIFO) logic (like a stack of plates).

49.3 Non-Linear Data Structures: Trees

Unlike arrays or linked lists, a tree is a hierarchical data structure. They are absolutely critical for database indexing.

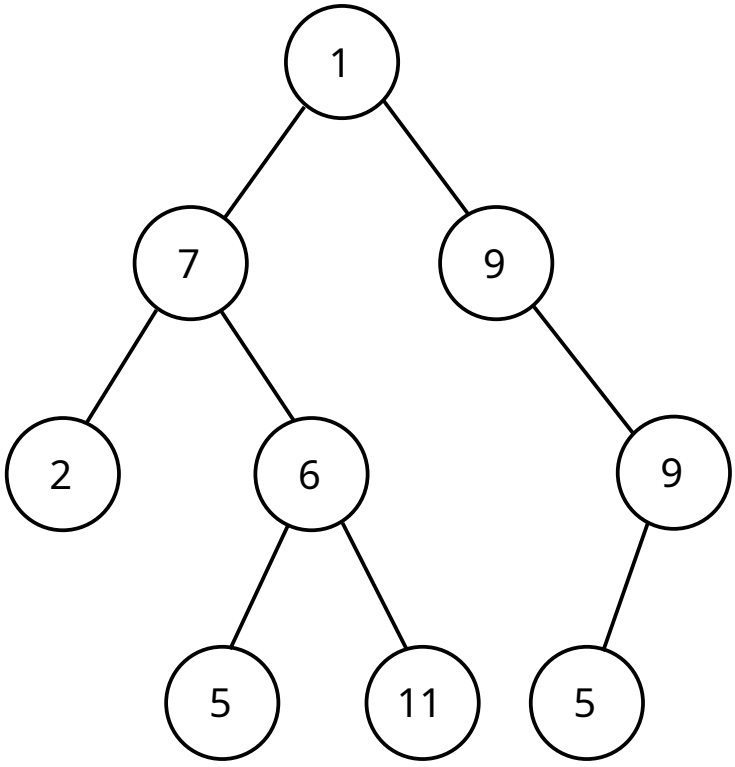


Figure 49.1: A visual representation of a binary tree structure.

Let's break down the terminology:

- **Root:** The topmost node.
- **Parent/Child:** A node connected to nodes directly below it is a parent.
- **Leaf Nodes:** Nodes with no children.
- **Arity (or Degree):** The number of children a specific node has.
- **Level / Depth:** The distance from the root.
- **Height:** The maximum level in the tree.

49.3.1 Binary Search Trees (BST)

A **Binary Search Tree** maintains a strict ordering property. For every node, all values in the **left subtree** are strictly *less* than the node's value, and all values in the **right subtree** are *greater*.

Let's build a BST from the data: `17, 22, 12, 10, 14, 25, 19`.

1. **Insert 17:** This is our first number, so it becomes the **Root**.
2. **Insert 22:** We compare 22 with 17. Since $22 > 17$, it goes to the **right**.
3. **Insert 12:** Compare 12 with 17. Since $12 < 17$, it goes to the **left**.
4. **Insert 10:** Compare 10 with 17 (left), then compare with 12. Since $10 < 12$, it goes to the **left** of 12. … and so on.

The final result is the beautiful structure shown below!

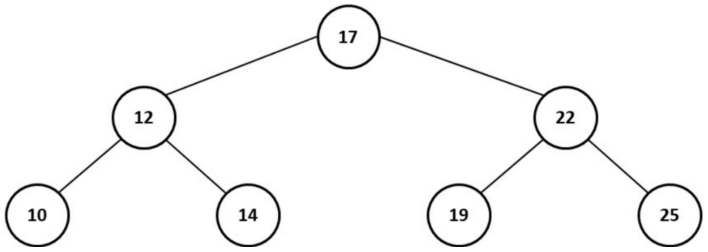


Figure 49.2: The fully constructed Binary Search Tree.

If the tree remains balanced, this property allows us to search, insert, and delete in $O(\log n)$ time, making it immensely faster than scanning an array.

49.4 Search and Sorting

- **Linear Search:** Sequential checking of every element. Complexity is $O(n)$.
- **Binary Search:** An extremely efficient search on **sorted arrays** (or Binary Search Trees). At each step, it halves the search space. Complexity is $O(\log n)$.
- **Sorting Algorithms:** Common algorithms to organize data include Quicksort ($O(n \log n)$), Mergesort ($O(n \log n)$), Heap Sort ($O(n \log n)$), and the much slower Insertion Sort ($O(n^2)$).

50 Physical Storage Systems

50.1 Physical Storage Media

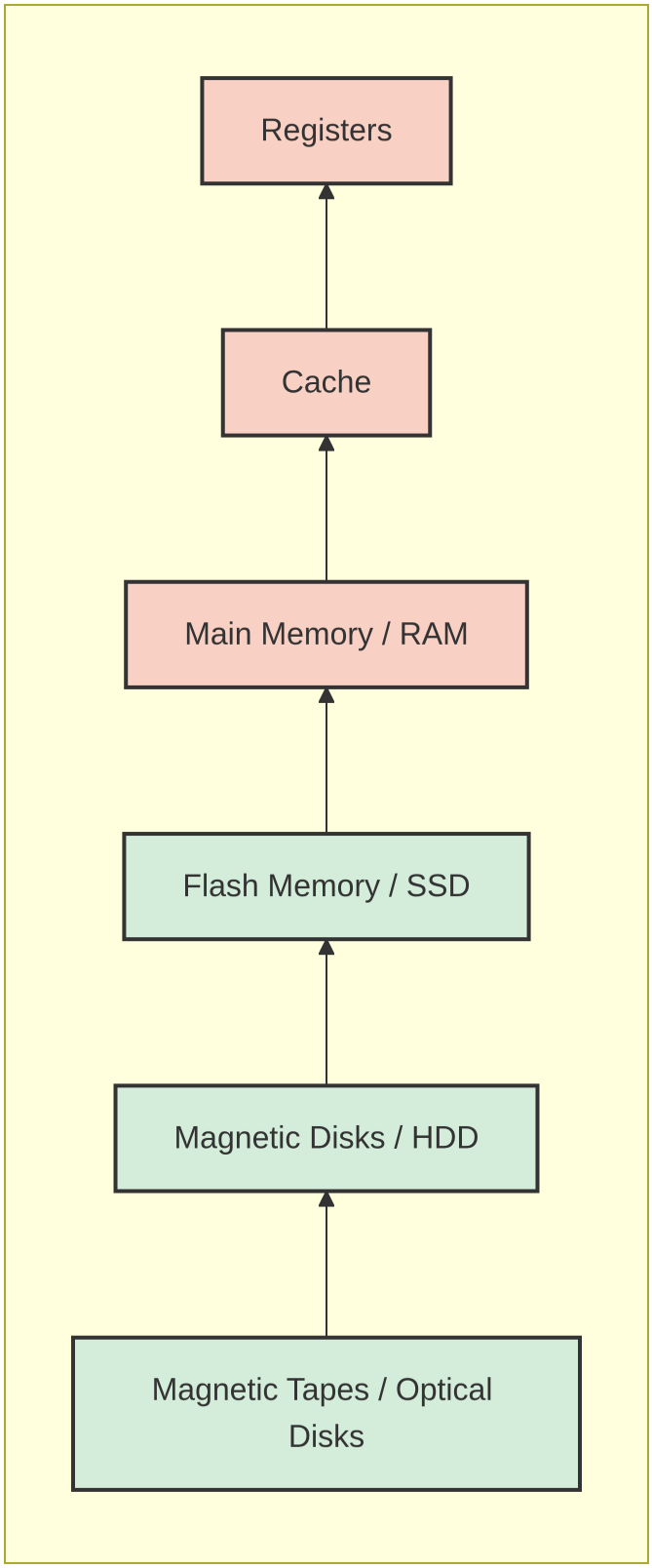
Data needs a place to live! Depending on how fast we need to access the data and how long we want it to survive without power, we use different types of storage media.

There is a fundamental divide in storage media: **Volatile** vs. **Non-Volatile**.

- **Volatile Storage:** Loses all data when the power is interrupted (e.g., Cache Memory, Main Memory / RAM). Extremely fast but temporary.
- **Non-Volatile Storage:** Retains data regardless of power status (e.g., Flash memory, Magnetic Disks, Optical Disks, Magnetic Tapes). Slower, but permanent.

50.1.1 The Storage Hierarchy Pyramid

In computer architecture, we make trade-offs. If you want **speed**, it costs more and you get **less capacity**. If you want **capacity**, it costs less per gigabyte, but you sacrifice **speed**.


Registers
Cache
Main Memory / RAM
Flash Memory / SSD
Magnetic Disks / HDD
Magnetic Tapes / Optical Disks

Note: As you move UP the pyramid, Speed and Cost per bit INCREASES. As you move DOWN the pyramid, Capacity and Access Time INCREASES.

50.2 Inside a Magnetic Disk (HDD)

To understand database performance, we must look at the physical mechanics of where our data actually lives: **Magnetic Disks**.

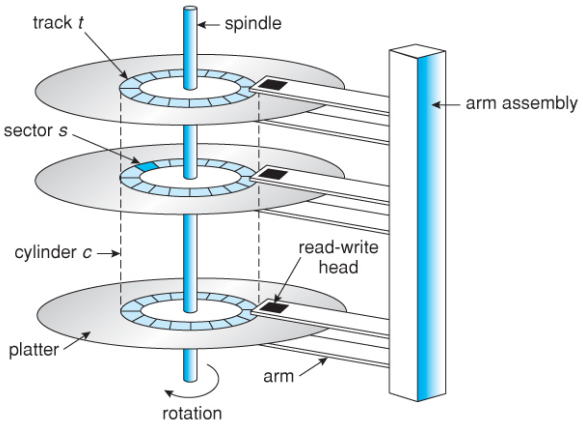


Figure 50.1: The inside of a typical magnetic disk.

Think of a hard drive like an old-school vinyl record player:

- **Platter (The Record):** The circular disk coated with magnetic material.
- **Spindle (The Turntable Motor):** The central rod that spins the platters at high speeds (e.g., 7200 RPM).
- **Read/Write Head (The Needle):** The tiny magnet that hovers above the platter.
- **Actuator Arm (The Tonearm):** The mechanical arm that moves the Read/Write head.

50.2.1 Data Organization on the Disk

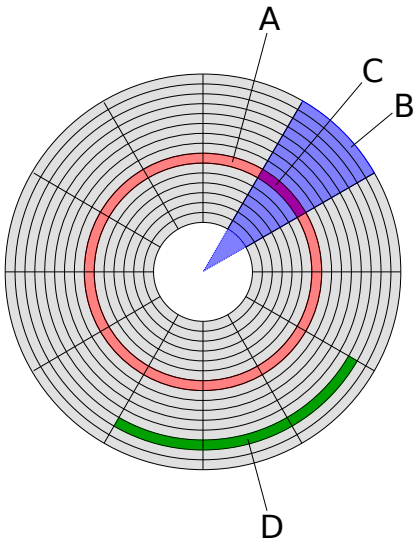


Figure 50.2: A breakdown of disk geometry.

- **(A) Track:** A single, concentric circular ring on the surface of the platter.
- **(C) Sector:** A track is divided into smaller arcs called sectors. It is the **smallest unit of data** that can be read or written (traditionally 512 bytes).
- **Cylinder:** The set of all tracks that are at the exact same position across all stacked platters.
- **(D) Block / Cluster:** A grouping of contiguous sectors used by the operating system to allocate space efficiently.

50.3 Disk Performance Metrics

When a database asks the hard drive for a piece of data, the mechanical parts have to move.

50.3.1 1. Access Time

The total time it takes to begin reading the actual data.

$$Access\ Time = Seek\ Time + Rotational\ Latency$$

- **Seek Time:** The time taken to physically reposition the arm over the correct track.
- **Rotational Latency:** The time spent waiting for the correct sector to rotate directly underneath the head.

50.3.2 2. Data Transfer Rate

The speed at which data is moved from the disk to the main memory once the head is in position.

50.3.3 3. Reliability (MTTF)

Hard drives are mechanical, which means they will eventually break. Reliability is measured by **Mean Time To Failure (MTTF)**. An MTTF of 5 years doesn't mean it will break in exactly 5 years, but it provides a statistical average of expected lifespan.

50.4 Emerging Technologies

While Magnetic Disks are still widely used, new technologies are constantly evolving storage:

- **Solid State Drives (SSD):** Use flash memory to store data persistently without any moving parts, significantly reducing Access Time.
- **Cloud Storage:** Abstracting the physical hardware entirely, relying on distributed remote servers.
- **DNA Digital Storage:** A futuristic approach using synthetic DNA to store massive amounts of data with extreme longevity.
- **Quantum Memory:** Theoretical storage systems leveraging quantum states for unprecedented speeds and security.

51 Disk Performance Math

51.1 Intuition Behind the Math

Memorizing formulas can be difficult. It is much easier if we understand *why* the formulas exist. Let's look at the basic math behind disk performance.

51.1.1 1. Capacity of a Disk

How much data can a disk hold? Imagine you have a bookshelf. The capacity of the shelf is the number of books it holds times the pages in one book. Similarly, a disk is made of sectors.

$$Capacity = Total\ No.\ of\ Sectors \times Size\ of\ One\ Sector$$

51.1.2 2. Time Period of One Rotation

If a disk spins at RPM (Revolutions Per Minute), how long does *one* single rotation take? If it does RPM revolutions in 60,000 milliseconds (1 minute), then one revolution takes:

$$Time\ Period = \frac{1\ minute}{RPM} = \frac{60,000\ ms}{RPM}$$

51.1.3 3. Transfer Rate

How much data can we read per unit of time? In one single rotation (Time Period), the head sweeps over an entire track. Therefore, the data transferred in one rotation is exactly the capacity of one track.

$$Transfer\ Rate = \frac{Bytes\ on\ one\ Track}{Time\ Period}$$

51.1.4 4. Transfer Time

If we know our speed (Transfer Rate) and the distance we need to cover (File Size), we can find the time! This is just $Time = Distance \div Speed$.

$$Transfer\ Time = \frac{File\ Size}{Transfer\ Rate}$$

51.1.5 5. Addressing Sectors

Every sector needs a unique ID (binary address) so the OS can find it. If you have 8 sectors, you need 3 bits to address them (000 to 111). This is because $2^3 = 8$. To find the number of bits required for N sectors, we take the base-2 logarithm.

$$Minimum\ Bits = \lceil \log_2(Total\ No.\ of\ Sectors) \rceil$$

(We use the ceiling function $\lceil \rceil$ because we can't have a fraction of a bit).

51.2 Solved Examples

Let's put our intuition to work with some structured numerical problems.

! Solved Example 1: Reading a File

Problem Statement: Consider you have a file named "IITM BSc" in your hard disk. The file size is 1000 KB. Seek time of your hard disk read head is 3ms, rotational speed is 30,000 RPM. The disk has 200 sectors/track and sector size is 512 bytes.

a) What is the transfer rate of your hard-disk (in KB/ms)?
b) Considering the fact that the file data is stored in all non-consecutive sectors, how much time will be required to read the whole file after the read request is made? (Note: Consider Access time + Transfer time)

Part A: Finding the Transfer Rate

1. **Given:**
   - $RPM = 30,000$
   - $Sectors\ per\ Track = 200$
   - $Sector\ Size = 512\ Bytes$
2. **Step-by-step Solution:**
   - First, find the Time Period for one rotation: $Time\ Period = \frac{60{,}000\ ms}{30{,}000} = 2\ ms$
   - Second, find the capacity of one track (Bytes on one track): $Track\ Capacity = 200\ sectors \times 512\ bytes = 102,400\ Bytes = 100\ KB$ *(Note: we assume* $1024\ Bytes = 1\ KB$*)*
   - Finally, calculate Transfer Rate: $Transfer\ Rate = \frac{100\ KB}{2\ ms} =$ **50 KB/ms**

Part B: Finding Total Read Time (Non-Consecutive Sectors)

1. **Given:**
    - $File\ Size = 1000\ KB$
    - $Seek\ Time = 3\ ms$
    - Data is stored in **non-consecutive** sectors. This is a critical detail! It means for *every single sector* we read, we have to suffer Seek Time and Rotational Latency!
2. **Step-by-step Solution:**
    - First, find the total number of sectors the file occupies: $File\ Size\ in\ Bytes = 1000\ KB \times 1024 = 1,024,000\ Bytes\ Number\ of\ Sectors = \frac{1{,}024{,}000\ Bytes}{512\ Bytes} = 2000\ Sectors$
    - Next, find the Access Time for *one* sector: $Average\ Rotational\ Latency = \frac{1}{2} \times Time\ Period = \frac{1}{2} \times 2\ ms = 1\ ms\ Access\ Time = Seek\ Time + Rotational\ Latency = 3\ ms + 1\ ms = 4\ ms$
    - Next, find the Transfer Time for *one* sector: $Transfer\ Time\ (1\ Sector) = \frac{Sector\ Size}{Transfer\ Rate} = \frac{0.5\ KB}{50\ KB/ms} = 0.01\ ms$
    - Finally, calculate the total time for all 2000 non-consecutive sectors: $Time\ per\ Sector = Access\ Time + Transfer\ Time = 4\ ms + 0.01\ ms = 4.01\ ms\ Total\ Time = 2000 \times 4.01\ ms =$ **8020 ms** (or 8.02 seconds)

! Solved Example 2: Disk Geometry and Addressing

Problem Statement: Consider a disk with 10 platters, 64 tracks/surface, 256 sectors/track, 512 bytes/sector. 4 bytes/sector is reserved for storing file system information (formatting data).

a) How much free space is available for use (in MB, upto two decimal places)?
b) How many bits are required for addressing all the sectors?

Part A: Finding Free Space

1. **Given:**
    - $Platters = 10$ (Note: A platter has 2 surfaces! So $Surfaces = 20$)
    - $Tracks\ per\ Surface = 64$
    - $Sectors\ per\ Track = 256$
    - $Sector\ Size = 512\ Bytes$
    - $Reserved\ Space = 4\ Bytes/sector$
2. **Step-by-step Solution:**
    - First, find the total number of sectors on the entire disk: $Total\ Sectors = Surfaces \times Tracks/Surface \times Sectors/Track\ Total\ Sectors = 20 \times 64 \times 256 = 327,680\ Sectors$

- Next, calculate the *usable* space per sector: $Usable\ Sector\ Size = 512 - 4 = 508\ Bytes$
- Finally, calculate the total free space: $Total\ Free\ Space = 327,680 \times 508\ Bytes = 166,461,440\ Bytes$ Convert to MB (divide by 1024^2): $Free\ Space\ in\ MB = \frac{166,461,440}{1,048,576} \approx \mathbf{158.75\ MB}$

Part B: Addressing Sectors

1. **Step-by-step Solution:**
   - We know there are $327,680$ total sectors on the disk.
   - We apply the addressing formula: $Minimum\ Bits = \lceil \log_2(327,680) \rceil$ Since $2^{18} = 262,144$ and $2^{19} = 524,288$, we know we need more than 18 bits. $Minimum\ Bits = \mathbf{19\ bits}$

52 File Organization & Buffering

52.1 File Structure and Organization

Before we can query a database, the raw data must be mapped onto the physical disk blocks we discussed in the previous chapters.

52.1.1 The Data Hierarchy

1. **Database:** The highest level, containing multiple files.
2. **File:** A collection of related records (e.g., the `Instructor` table).
3. **Record:** A single instance of data (e.g., a specific instructor's row).
4. **Field / Attribute:** The individual data points within a record (e.g., Name, Salary).

52.1.2 Record Management

- **Fixed-length Records:** Every record takes up the exact same amount of space. This makes math easy, but deletions leave "holes". These are managed via periodic compaction, shifting records, or using a "Free List" to track empty spaces.
- **Variable-length Records:** Used when attributes vary in size (like `VARCHAR`). These are much harder to manage and usually require storing offsets and using **Slotted Pages** to organize the data block dynamically.

52.1.3 File Organization Types

How do we arrange these records inside a file?

- **Heap Organization:** Records are placed wherever there is free space. Completely unordered.
- **Sequential Organization:** Records are strictly ordered by a "Search Key" (like ordering a dictionary alphabetically). Excellent for sequential processing.

- **Multi-table Clustering:** Purposely grouping related records from *different* tables (e.g., a `Department` record physically stored right next to all its `Instructor` records) into the same disk block. This minimizes expensive disk I/O when performing SQL `JOIN` operations.

*Note: The **Data Dictionary (System Catalog)** is a special file that stores metadata—data about the data itself! It tracks schemas, users, and integrity constraints.*

52.2 Storage Access and Buffer Management

We've established a crucial fact: **Disks are slow, but Main Memory (RAM) is fast.** Every time the database needs to read or write a record, it must transfer an entire **Block** of data between the disk and main memory. This disk I/O is the biggest bottleneck in database performance.

Enter the **Buffer Manager**.

52.2.1 The Role of the Buffer Manager

The Buffer Manager is a software subsystem responsible for allocating space in main memory to temporarily hold blocks fetched from the disk.

When a query requests data:

1. The Buffer Manager checks if the required block is already in the buffer (Main Memory).
2. If **YES**, this is a "Cache Hit". The data is served instantly.
3. If **NO**, this is a "Cache Miss". The Buffer Manager must fetch the block from the slow magnetic disk.

52.2.2 Replacement Policies

If the buffer is completely full of blocks, and we need to fetch a new one from the disk, which old block should we evict (throw out) to make room? This is determined by a **Replacement Policy**.

Least Recently Used (LRU)

The most common strategy. The Buffer Manager evicts the block that hasn't been accessed for the longest time.
Intuition: If you haven't looked at a block in a long time, you probably won't need it again soon.

Most Recently Used (MRU)

Sometimes LRU is a terrible idea! Consider a query using a **Nested Loop Join** where we repeatedly scan the same large relation (e.g., scanning the entire `Instructor` table for *every single row* in the `Department` table).
If the `Instructor` table is larger than our buffer, LRU will evict the beginning of the table just before we need to scan it again! In this specific scenario, **Most Recently Used (MRU)** is better. We evict the block we *just* finished reading, because in a looping scan, we won't need it again until the next full loop.

52.2.3 Pinned Blocks

Some operations are so critical that a block cannot be evicted under any circumstances, even if it's the LRU or MRU. For example, if a block is currently being actively updated by a transaction, the Buffer Manager will **Pin** the block, restricting it from being written back to disk prematurely.

Part IX

Week-09

53 Indexing

53.1 What is an Index?

Think of a database index like the index at the back of a textbook — instead of reading every page, you look up the term, get the page number, and jump straight there.

An index is a smaller, separate data structure that maps **search-key values** to their records in the main table. Each entry in an index file is called an **index entry** and looks like this:

search-key	pointer
"Pabitra Mitra"	→ record #6

There are two broad families:

- **Ordered indices** — entries are sorted on the search key.
- **Hash indices** — entries are scattered into buckets using a hash function. Fast for exact lookups, but can't handle range queries.

This section focuses on ordered indices.

53.2 Ordered Indices: Primary vs. Secondary

Primary index (clustering index): The index whose search key matches the physical order of the data file. Efficient for sequential scans since adjacent records on disk are also logically adjacent.

Secondary index (non-clustering index): An index on a different attribute than the one the file is sorted on. Pointers are scattered across the disk, so sequential scans are expensive.

> A table can have **only one primary index** but **many secondary indices**.

An **index-sequential file** is a sequentially ordered file paired with its primary index, also called ISAM (Indexed Sequential Access Method).

53.3 Dense vs. Sparse Indexes

53.3.1 Dense Index

One index entry for **every record** in the data file.

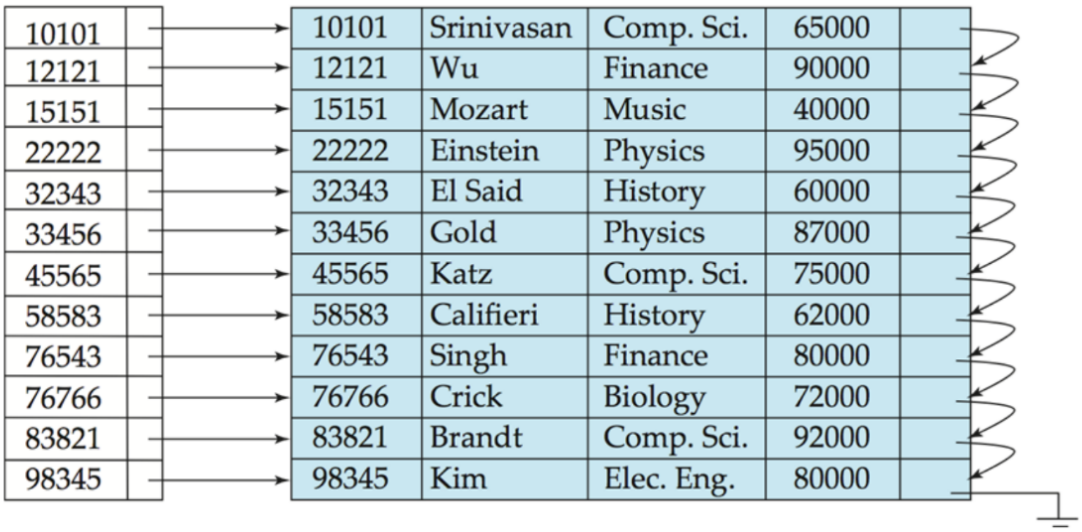


Figure 53.1: Dense Index

Advantage: Can check record existence from the index alone.
Disadvantage: Index grows proportionally with the data.

53.3.2 Sparse Index

One index entry per **disk block** (not per record).

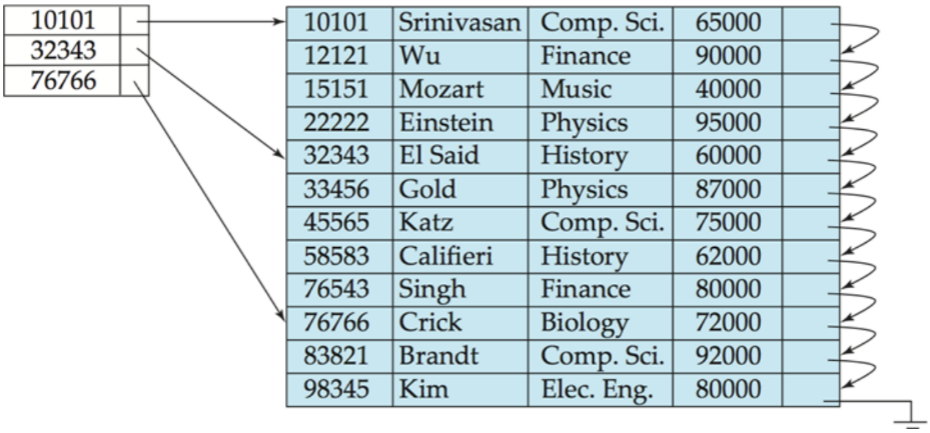


Figure 53.2: Sparse Index

To find key K: find the largest index entry $\leq K$, follow its pointer to the block, then scan within the block.

Advantage: Much smaller index, less maintenance.
Disadvantage: Slightly slower — requires a sequential scan within the block.

> One sparse index entry per disk block is the standard trade-off — small enough for memory, granular enough to avoid long scans.

Secondary indices must be dense because records with a given secondary key are scattered across the file — a sparse index would miss records not sitting at a block boundary.

53.4 Multilevel Index

Problem: If the primary index itself is too large to fit in memory, searching it becomes expensive.

Solution: Build a sparse index *on top of* the primary index.

- **Inner index** — the primary index file (on disk).
- **Outer index** — a sparse index of the inner index (fits in memory).

Lookup path: outer index (memory) → inner index block → data record.

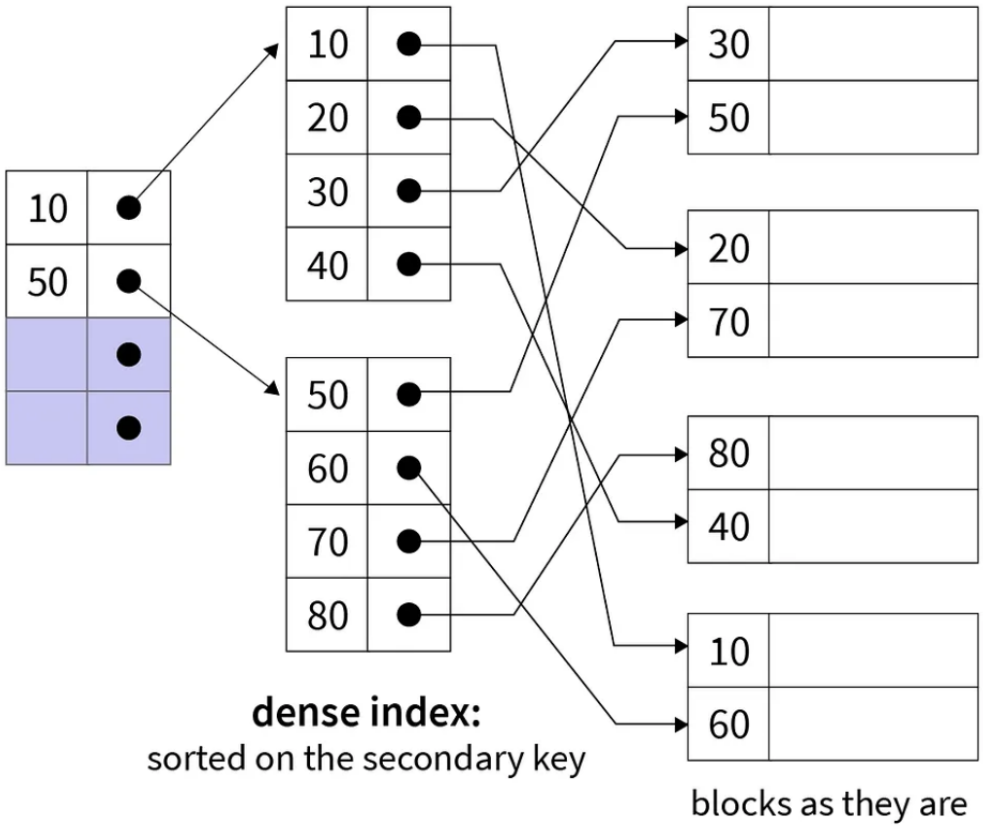


Figure 53.3: Multilevel Index

If even the outer index is too large, add another level. This layering idea is the foundation of B-trees.

53.5 Index Updates

53.5.1 On Deletion

- If the deleted record was the **only** one with that key value, remove the index entry.

- **Dense:** treat like a sorted-list deletion.
- **Sparse:** replace the entry with the next key in the block; if the next key already has its own entry, just delete the current one.

53.5.2 On Insertion

- **Dense:** if the key is absent from the index, insert a new entry in sorted order.
- **Sparse:** no change needed unless a new disk block is created — if so, add an entry for the first key in that new block.

Multilevel updates follow the same logic, applied at each level recursively.

53.6 Index Definition in SQL

```
-- Create a basic index
CREATE INDEX emp_ename ON emp_tab(ename);

-- Create a unique index (enforces candidate key constraint)
CREATE UNIQUE INDEX emp_email ON emp_tab(email);

-- Create a composite index on two columns
CREATE INDEX emp_comp ON emp_tab(ename, empno) COMPUTE STATISTICS;

-- Function-based index (precomputes the function result for each row)
CREATE INDEX emp_upper ON emp_tab(UPPER(ename)) COMPUTE STATISTICS;

-- Bitmap index (best for columns with few distinct values)
CREATE BITMAP INDEX idx_gender ON student(gender);

-- Drop an index
DROP INDEX emp_ename;
```

> `COMPUTE STATISTICS` tells the optimizer to gather column statistics at index creation time — important for the optimizer to make good decisions.
>
> `CREATE UNIQUE INDEX` is technically redundant if you've already declared the column as `UNIQUE` in the table definition — the constraint is preferred.

53.7 Guidelines for Indexing

Indexing is an ongoing task — query patterns change, tables grow, and indexes need to be monitored and adjusted over time.

- *Rule 1*: Index the Correct Tables
- *Rule 2*: Index the Correct Columns
- *Rule 3*: Limit the Number of Indexes for Each Table
- *Rule 4*: Choose the Order of Columns in Composite Indexes
- *Rule 5*: Gather Statistics to Make Index Usage More Accurate
- *Rule 6*: Drop Indexes That Are No Longer Required

53.8 Formulas for Solving the Problem

$$\text{Space of Index} = \text{Size of Key Field} + \text{Size of Pointer Field}$$

$$\text{Total Index Space} = \text{Space of Index} \times \text{No. of Records}$$

$$\text{One Block Size} = \frac{\text{Size of File}}{\text{No. of Blocks Used to Store the File}}$$

$$\text{No. of Blocks for Index} = \left\lceil \frac{\text{Total Index Space}}{\text{One Block Size}} \right\rceil$$

Always round **up** — a partial block still occupies a full block on disk.

54 Tree-Based Index Structures

54.1 Why Trees for Indexing?

The multilevel index from Section 1 was a good idea, but it had one major problem: as the file grows, overflow blocks pile up and performance degrades. You eventually need to reorganize the whole file — expensive and disruptive.

Tree-based structures solve this by **self-balancing on every insert and delete**, keeping search cost at $O(\log n)$ always, without any manual reorganization.

54.2 The 2-3-4 Tree: Motivation

The problem with BSTs on disk is that each **node holds only one key and has only two children**. The tree must be very tall ($O(\log n)$ levels) to hold n records.

What if we could make each node **fatter** — storing multiple keys and having multiple children? Then the tree would be much **shorter**, and we'd need far fewer disk reads to reach any record. This idea leads to the **2-3-4 Tree**.

54.3 What Is a 2-3-4 Tree?

A 2-3-4 Tree is a balanced search tree where every node can hold **1, 2, or 3 keys** (and correspondingly have **2, 3, or 4 children**). That is where the name comes from.

The three node types:

Node type	Keys stored	Children	Search key rule
2-node	1 (call it S)	2	left child $<$ S; right child $>$ S
3-node	2 (S and L)	3	left $<$ S; middle: S–L; right $>$ L
4-node	3 (S, M, L)	4	left $<$ S; second: S–M; third: M–L; right $>$ L

A leaf node can hold 1, 2, or 3 data items.

Key properties:

- All leaves sit at the **same depth** — the tree is always perfectly height-balanced.
- $h \sim O(\log n)$, so search, insert, and delete all cost $O(\log n)$.
- Data is always kept in **sorted order**.

54.4 Search in a 2-3-4 Tree

Search is a straightforward extension of BST search. At each node, compare the target key against the node's keys and follow the appropriate child pointer. Repeat until you reach a leaf or find the key.

> Because all leaves are at the same depth, every search takes exactly $O(h) = O(\log n)$ steps — no worst-case skew possible.

54.5 Insertion in a 2-3-4 Tree

Insertion always happens at a **leaf**. The process:

1. Search for the expected location (the leaf where the key would go).

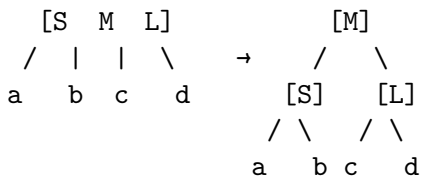
2. If the leaf is a **2-node**, expand it to a **3-node** and insert. Done.
3. If the leaf is a **3-node**, expand it to a **4-node** and insert. Done.
4. If the leaf is a **4-node**, we can't add another key — we must **split** it first.

54.5.1 Node Splitting

Splitting a 4-node takes its **middle key (M)** and pushes it up to the parent. The original 4-node becomes two 2-nodes (one holding S, one holding L), both children of the parent which now absorbs M.

There are three cases depending on the parent:

Case 1 — Parent is the root (a 4-node):



The tree grows one level taller.

Case 2 — Parent is a 2-node:

The middle key M is absorbed into the parent, converting it from a 2-node to a 3-node. The split 4-node's two halves (S and L) become two separate children.

Case 3 — Parent is a 3-node:

Same idea — M is absorbed, converting the parent from a 3-node to a 4-node.

54.5.2 Early vs. Late Splitting

There are two strategies for *when* to split:

- **Early splitting:** Split any 4-node you *encounter* on the way down during a search, before you actually need to. This ensures you never have a full node at the insertion point. At most one split per level.
- **Late splitting:** Only split when you actually need to insert into a 4-node. May cascade up $O(h)$ splits in the worst case.

Both strategies produce valid trees with the same complexity $O(h)$, but they produce different final shapes. The slides follow the **early splitting** strategy.

54.6 Insertion Example: Step by Step

Inserting the sequence **10, 30, 60, 20, 50, 40, 70, 80, 15, 90, 100** using the **early splitting** strategy.

After inserting 10, 30, 60 — all three fit into one leaf (a 4-node):

```
[10  30  60]
```

Insert 20 — the root is a 4-node, so we split it *before* descending. Middle key 30 becomes the new root; 10 and 60 become separate children. Then 20 is inserted into the left child:

```
      [30]
     /    \
[10, 20]  [60]
```

Insert 50 — $50 > 30$, go right. [60] is a 2-node, so just expand it:

```
      [30]
     /    \
[10, 20]  [50, 60]
```

Insert 40 — $40 > 30$, go right. [50, 60] is a 3-node, expand to 4-node and insert:

```
      [30]
     /    \
[10, 20]  [40, 50, 60]
```

Insert 70 — root [30] is fine. $70 > 30$, descend right. [40, 50, 60] is a 4-node — split it immediately (early strategy). Middle key 50 rises to join the root:

```
          [30, 50]
         /   |    \
[10,20]  [40]  [60]
```

Now insert 70 into the right child [60]:

```
          [30, 50]
         /   |    \
[10,20]  [40]  [60, 70]
```

Insert 80 — $80 > 50$, go to right child [60, 70]. It's a 3-node, expand it:

```
          [30, 50]
         /   |    \
[10,20]  [40]  [60, 70, 80]
```

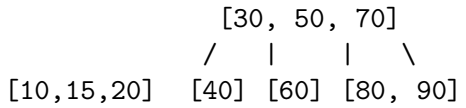
Insert 15 — $15 < 30$, go left. [10, 20] is a 3-node, expand it:

```
             [30, 50]
            /   |    \
[10, 15, 20]  [40]  [60, 70, 80]
```

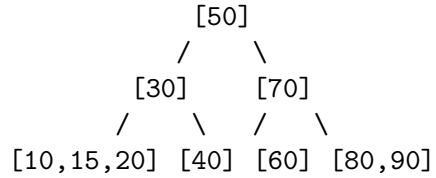
Insert 90 — root [30, 50] is fine. $90 > 50$, descend right. [60, 70, 80] is a 4-node — split it. Middle key 70 rises to join the root:

```
               [30, 50, 70]
              /   |    |   \
[10,15,20]  [40] [60] [80]
```

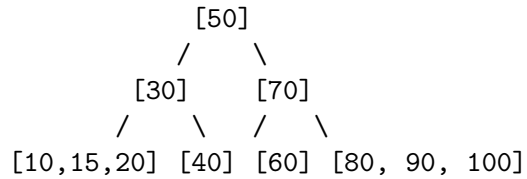
Now insert 90 into the rightmost child [80]:



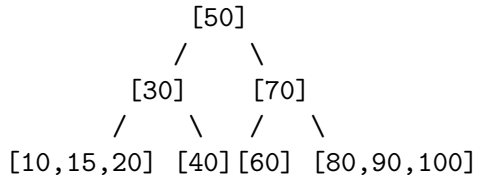
Insert 100 — root [30, 50, 70] is a 4-node — split it first. Middle key 50 becomes the new root; 30 and 70 become children:



Now insert 100. $100 > 50 \rightarrow$ go right to [70]. $100 > 70 \rightarrow$ go to right child [80, 90]. It's a 3-node, expand it:



Final tree after all 11 insertions:



Height = 2. All 11 keys stored, all leaves at the same level.

54.7 Deletion in a 2-3-4 Tree

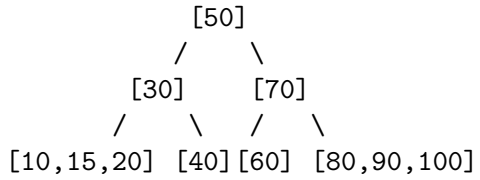
Deletion is the reverse of insertion. The core idea:

1. Locate the node containing the target key.
2. Swap it with its **inorder successor** (the smallest key in the right subtree). This reduces deletion to always happening at a **leaf**.
3. If the leaf is a 3-node or 4-node, simply remove the key.
4. If the leaf is a 2-node, removing its only key would leave an empty node — which is illegal. To prevent this, **transform every 2-node you encounter** (on the way down) into a 3-node or 4-node first. This is the reverse of the splitting cases from insertion.

54.7.1 Deletion Example

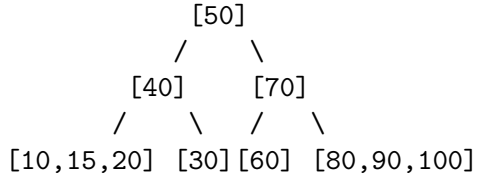
We will delete **30** and then **50** from the final tree above.

Starting tree:



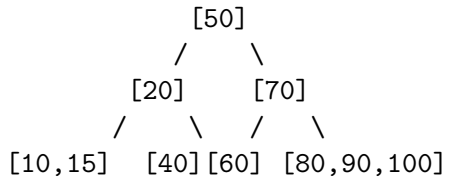
Delete 30

30 is an internal node. Its inorder successor is **40** (smallest key in its right subtree). Swap 30 and 40:



Now delete 30 from the leaf [30]. That leaf is a 2-node — but since 30 is the only key, removing it would leave an empty node.

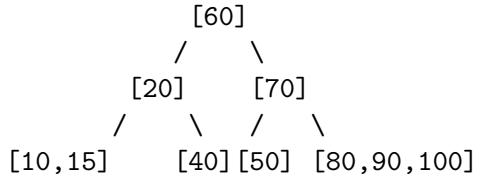
However, looking at its sibling [10, 15, 20] (a 4-node), we can **borrow** from it. The sibling's largest key (20) rotates up to the parent, and the parent's key (40) rotates down to fill the now-empty spot:



30 is deleted. The tree remains balanced and all leaves are still at the same depth.

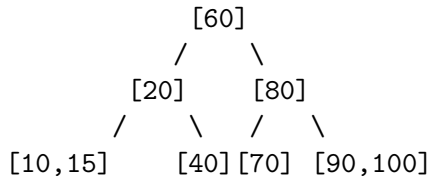
Delete 50

50 is the root. Its inorder successor is **60** (smallest key in its right subtree). Swap 50 and 60:

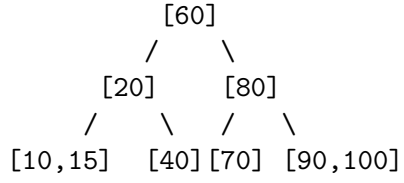


Now delete 50 from the leaf [50]. It is a 2-node with no siblings to borrow from (both [20] and [60] in the right subtree are also 2-nodes).

In this case, we **merge**: combine the 2-node [50] with its sibling [60] (oops — actually [60] is now the root, not a sibling). Let's be precise: the right child of the root is [70], whose left child is [50] and right child is [80,90,100]. Since [50] is a 2-node and we must delete from it, we can borrow from the rich sibling [80, 90, 100]. The sibling's smallest key (80) rotates up to the parent [70], and 70 rotates down to fill the gap:



50 is deleted. Final tree:



Still height 2, all leaves at the same depth.

> **Key takeaway from deletion:** The 2-3-4 tree never lets a node go empty. If you need to delete from a 2-node, you always first fatten it up — either by borrowing from a sibling or by merging with a sibling — before removing the key. This is the structural guarantee that keeps the tree balanced after every operation.

54.8 B+ Trees

ISAM (indexed-sequential files) degrades as the file grows — overflow blocks pile up and require periodic full reorganization. B+ trees fix this by **self-balancing on every insert and delete**.

With $n = 100$ and 1 million records, B+ Tree needs 4 reads. A balanced BST on the same data needs ~20 disk reads. Each read costs ~20ms — the difference is significant at scale.

54.9 Structure

A B+ tree with parameter n:

Node	Min	Max
Internal (non-root)	$\lceil n/2 \rceil$ children	n children
Root (non-leaf)	2 children	n children
Leaf	$\lceil (n-1)/2 \rceil$ values	$n-1$ values
Root as leaf	0 values	$n-1$ values

- All paths root $\rightarrow$ leaf are the **same length**.
- Keys in each node are ordered: $K_1 \leq K_2 \leq \cdots \leq K_{n-1}$.
- Leaf nodes are **linked** $\rightarrow$ supports sequential (range) access.
- **Leaf:** $P_i \rightarrow$ record with key K_i; last pointer $P_n \rightarrow$ next leaf.
- **Internal:** $P_i \rightarrow$ subtree. Keys in $P_1 < K_1$; keys in $P_i \geq K_{i-1}$ and $< K_i$; keys in $P_n \geq K_{n-1}$.

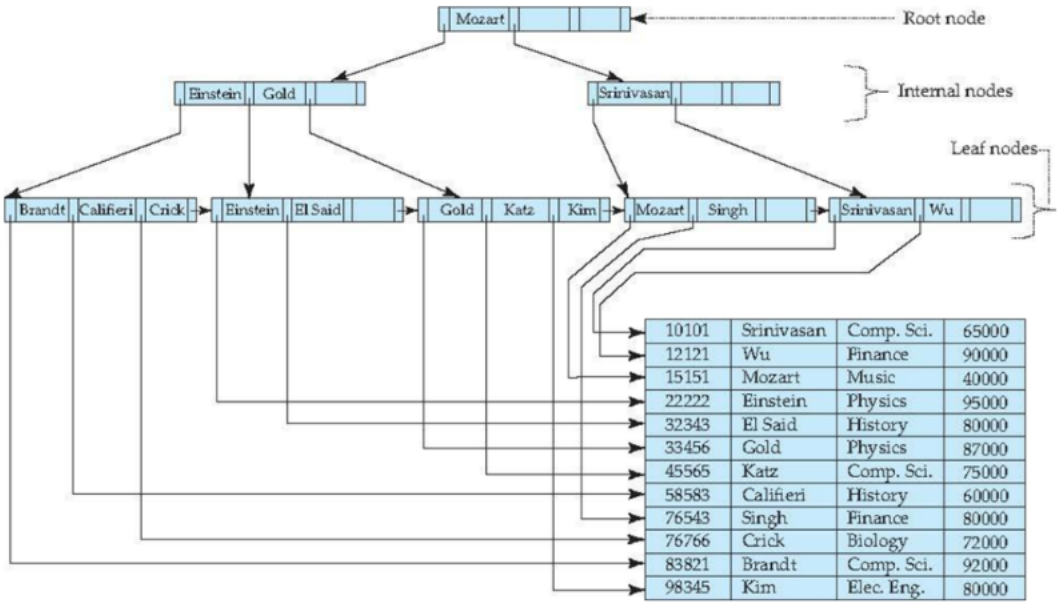


Figure 54.1: B+Tree

54.10 Search

To find value V:

1. Start at the root.
2. At each internal node, scan the keys left to right and find where fits. Follow the pointer to the matching child subtree. Repeat until you reach a leaf.
3. At the leaf, scan for . If found, follow its pointer to the record. If not, the record doesn't exist.

Duplicates: ordering relaxes to $\leq$. Always descend P_i when $V = K_i$ — duplicates may exist in the left subtree. To find all occurrences, traverse consecutive leaves via the linked list.

54.11 Insertion

1. Find the target leaf and insert in sorted order.
2. **If space exists** → insert. Done.
3. **If leaf is full** → split:
   - Put first $\lfloor n/2 \rfloor$ pairs in original node, rest in new node.
   - Push smallest key of new node up to parent.
4. If parent is also full → split it and propagate upward.
5. Worst case: root splits, tree grows one level taller.

54.12 Deletion

1. Remove the record and its (key, pointer) from the leaf.
2. **Node still has enough entries** → done.
3. **Underfull + can merge with sibling** → combine into one node; pull separator key down from parent; recurse upward.
4. **Underfull + cannot merge** → redistribute from sibling; update separator key in parent.
5. If root ends up with one child → delete root, promote that child.

54.13 B-Tree vs B+ Tree

B-tree eliminates key duplication — each key appears **once** across the whole tree, with record pointers directly in internal nodes.

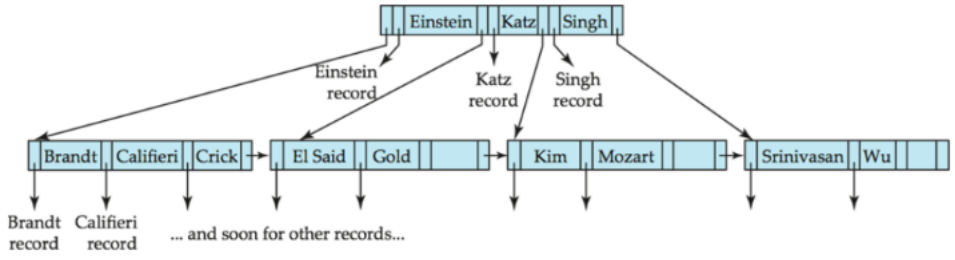


Figure 54.2: B Tree

	B+ Tree	B-Tree
Keys in internal nodes	Routing copies only	Actual record pointers
Key duplication	Yes	No
Leaf nodes linked	Yes	No
Fan-out	Higher	Lower
Sequential access	Efficient	Inefficient
Early key find	No	Sometimes
Complexity	Simple	More complex

55 Hashing and Bitmap Indices

55.1 What is Hashing?

Instead of searching through a sorted structure, hashing computes **exactly which bucket** a record lives in — no traversal needed. Think of it as a direct address calculator.

A **hash function** h maps a search key to a bucket address:

$$h : \text{search-key values} \rightarrow \text{bucket addresses}$$

- A **bucket** is a unit of storage — typically one disk block — holding one or more records.
- If $h(k_1) = h(k_2)$ for $k_1 \neq k_2$, a **collision** has occurred.

A good hash function is **uniform** (distributes keys evenly across all buckets) and **fast** to compute.

> The worst hash function maps every key to the same bucket — search degrades to $O(n)$.

55.2 Static Hashing

The number of buckets B is **fixed** at creation time.

Lookup: compute $h(k) \rightarrow$ go directly to that bucket $\rightarrow$ scan within it sequentially.

Example: 10 buckets, keys are department names. Hash = sum of character values mod 10.

```
h(Music)      = 1    h(History)   = 2
h(Physics)    = 3    h(Elec. Eng.) = 3   ← collision
```

Physics and Elec. Eng. both map to bucket 3. Both records are stored there and the bucket is scanned sequentially to find the right one.

55.3 Handling Bucket Overflow

Overflow happens when a bucket fills up. Causes:

- Too few buckets allocated initially.
- **Skew** — many records share the same key, or the hash function distributes unevenly.

Solution: overflow chaining — link overflow buckets to the original in a linked list.

```
Bucket 3: [Physics | Elec. Eng.] → [overflow block] → ...
```

This scheme is called **closed hashing**.

55.4 Deficiencies of Static Hashing

The fixed bucket count B is the core problem:

Scenario	Problem
File grows beyond initial estimate	Too many overflow chains → poor performance
File shrinks	Mostly empty buckets → wasted space
B set too large from the start	Space wasted immediately

Periodic full reorganization with a new hash function is possible but expensive and disruptive.

Better fix: let the number of buckets grow and shrink dynamically.

55.5 Dynamic Hashing (Extendable Hashing)

Instead of a fixed bucket count, extendable hashing uses a **directory** (bucket address table) that doubles or halves as needed.

55.5.1 Core Idea

- The hash function produces a b-**bit integer** (typically $b = 32$).
- Only the **first** i **bits** (global depth) are used to index into the directory.
- Directory size $= 2^i$. Initially $i = 0$.
- Each bucket j stores its own **local depth** i_j — how many bits actually identify its entries.
- Multiple directory entries can point to the **same bucket** when $i_j < i$.

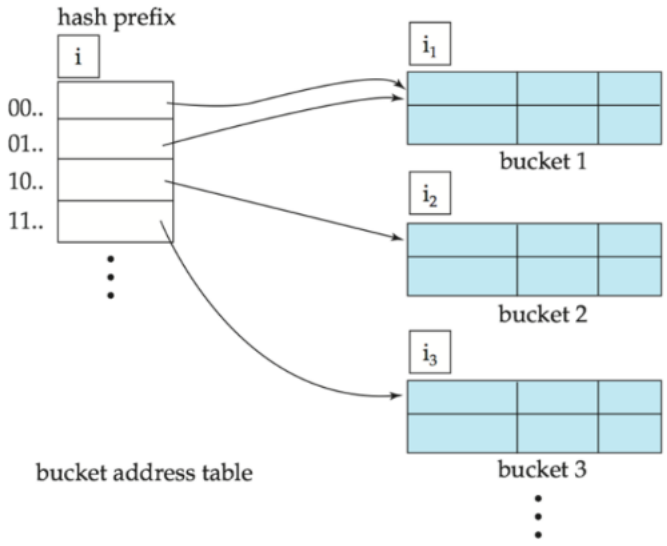


Figure 55.1: Dynamic Hashing

55.5.2 Lookup

1. Compute $h(k) = X$.
2. Use the first i bits of X as a directory index.
3. Follow the pointer to the bucket and scan within it.

55.5.3 Insertion

1. Lookup the target bucket j.
2. If space exists $\rightarrow$ insert. Done.
3. If bucket j is full $\rightarrow$ **split**:

Case 1 — $i > i_j$ **(multiple entries point to j):** - Allocate new bucket z, set $i_j = i_z = i_j + 1$. - Redirect the second half of entries that pointed to $j \rightarrow$ now point to z. - Redistribute all records in j between j and z by recomputing hash prefixes. - Retry insertion.

Case 2 — $i = i_j$ **(only one entry points to j):** - Increment i, double the directory (each old entry becomes two entries pointing to the same bucket). - Now $i > i_j \rightarrow$ apply Case 1. - If i hits limit b or too many splits occur, use an overflow bucket instead.

55.5.4 Deletion

1. Remove the record from its bucket.
2. If the bucket empties, it can be removed.
3. **Coalesce** with a "buddy" bucket — a bucket with the same i_j and the same $(i_j - 1)$-bit prefix.
4. Shrink the directory if the actual bucket count becomes much smaller than 2^i (expensive — do sparingly).

55.5.5 Extendable Hashing: Pros and Cons

Advantages	Disadvantages
Performance doesn't degrade as file grows	Extra level of indirection on every lookup
Minimal space overhead	Directory can exceed memory size
No periodic reorganization needed	Doubling the directory is expensive

If the directory outgrows memory, a B+ tree can index into it — combining the benefits of both structures.

55.6 Bitmap Indices

Bitmap indices are designed for **multi-attribute analytical queries** on low-cardinality columns — not for point lookups.

55.6.1 What They Are

For an attribute with m distinct values, a bitmap index stores m **bit vectors** — one per distinct value. Each bit vector has one bit per record:

- **1** if the record has that value.
- **0** otherwise.

Example: `gender` attribute, 5 records:

```
Record:     0  1  2  3  4
Gender:     M  F  M  F  M

Bitmap(M):  1  0  1  0  1
Bitmap(F):  0  1  0  1  0
```

> Best suited for columns with **few distinct values** — gender, country, income bracket, status flags. Not useful for high-cardinality columns like names or IDs.

55.6.2 Answering Queries with Bitmaps

Multi-attribute queries use **bitwise operations** — extremely fast at the CPU level.

Operation	Meaning
AND	Records matching both conditions
OR	Records matching either condition
NOT	Records not matching a condition

Example: find male instructors at income level L1:

```
Bitmap(gender=M):    1 0 0 1 0
Bitmap(income=L1):   1 0 1 1 0
                     ---------
AND result:          1 0 0 1 0   → records 0 and 3 match
```

Part X

Week-10

56 Transactions and ACID Properties

56.1 Introduction to Transactions

When you use a database, you don't typically run just a single isolated query. Often, you need to run a sequence of queries to accomplish a logical task.

A **Transaction** is a unit of program execution that accesses and, possibly, updates various data items. It represents a single logical unit of work.

56.1.1 The Big Challenges

When managing transactions, databases have to overcome two massive hurdles:

1. **System Failures:** What happens if the hardware crashes right in the middle of a transaction?
2. **Concurrent Execution:** What happens if two different transactions try to update the exact same row of data at the exact same millisecond?

56.2 The ACID Properties

To solve these challenges and guarantee database reliability, every transaction must adhere to four strict rules, famously known by the acronym **ACID**.

56.2.1 1. Atomicity (All or Nothing)

A transaction is treated as a single, indivisible unit. Either *all* of its instructions execute successfully, or *none* of them do. There is no such thing as a partially completed transaction.

56.2.2 2. Consistency

Execution of a transaction in isolation preserves the consistency of the database. If the database was in a valid state before the transaction, it must be in a valid state after the transaction completes.

56.2.3 3. Isolation

Even though transactions may be executing concurrently, the system guarantees that each transaction is unaware of the others. The end result must be equivalent to if the transactions had run sequentially, one after the other.

56.2.4 4. Durability

Once a transaction has committed successfully, its changes to the database are permanent, even if the system crashes immediately afterward.

56.3 A Classic Example: Bank Transfer

Let's look at the textbook example of transferring $50 from Account A to Account B. This is a single logical transaction, but it consists of several steps.

Step	Instruction	Meaning
1	`read(A)`	Get the current balance of Account A.
2	`A := A - 50`	Deduct $50 from A's balance.
3	`write(A)`	Save the new balance of A to the database.
4	`read(B)`	Get the current balance of Account B.
5	`B := B + 50`	Add $50 to B's balance.
6	`write(B)`	Save the new balance of B to the database.

Let's evaluate this against our ACID properties:

- **Atomicity:** Imagine the system crashes precisely after Step 3. Account A has lost $50, but Account B hasn't received it! The $50 just vanished. Atomicity ensures that if it fails here, the database will automatically roll back the changes to Account A, as if the transaction never started.

- **Consistency:** The sum of A + B must remain exactly the same before and after the transaction.
- **Isolation:** If another transaction tries to read Account A at Step 4, it might read the partially updated state. Isolation ensures this intermediate state is hidden.

56.4 Transaction States

As a transaction progresses from start to finish, it passes through several defined states.

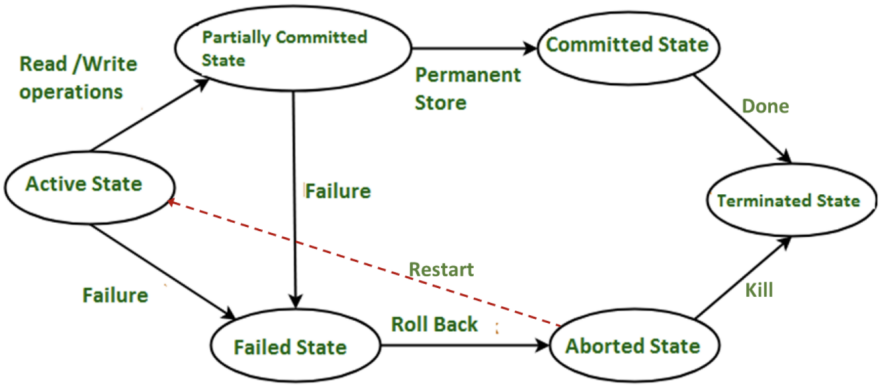


Figure 56.1: The lifecycle states of a transaction.

1. **Active:** The initial state. The transaction stays in this state while it is executing.
2. **Partially Committed:** After the final statement has been executed, but before the changes have been permanently recorded.
3. **Committed:** After successful completion. The changes are now durable and permanent.
4. **Failed:** Occurs when normal execution can no longer proceed (e.g., due to a software error, constraint violation, or hardware crash).
5. **Aborted:** After the transaction has been rolled back and the database has been restored to its prior state. From here, the transaction can either be restarted or killed entirely.

57 Schedules and Serializability

57.1 What is a Schedule?

When multiple transactions are executed concurrently, their instructions are interleaved. A **Schedule** is simply a sequence of instructions that specifies the chronological order in which the instructions of concurrent transactions are executed.

- A schedule must consist of *all* instructions from the participating transactions.
- It must preserve the order in which the instructions appeared in their original transactions.

57.1.1 Serial vs. Concurrent Schedules

- **Serial Schedule:** Transactions run one after the other. Transaction A finishes completely before Transaction B begins. This guarantees database consistency, but it is extremely slow.
- **Concurrent Schedule:** Instructions from multiple transactions are interleaved. This improves performance and resource utilization, but risks creating an inconsistent database state.

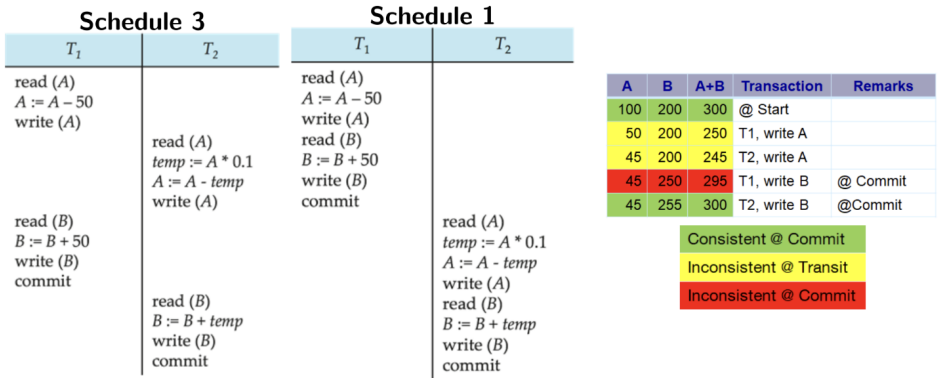


Figure 57.1: An example showing how an interleaved concurrent schedule can be equivalent to a serial schedule.

57.2 Serializability

A concurrent schedule is considered **Serializable** if its final outcome is identical to that of some purely serial schedule. If we can prove a schedule is serializable, we guarantee it preserves database consistency!

There are two primary ways to determine this: **Conflict Serializability** and **View Serializability**.

57.2.1 1. Conflict Serializability

To understand conflict serializability, we first need to define what a "conflict" is.

> **Conflicting Instructions**
>
> Let I_i and I_j be two instructions from different transactions, operating on the *exact same* data item Q (e.g. a column in a table, like `student_name`).
>
> - $read(Q)$ and $read(Q)$ → **NO CONFLICT**
> - $read(Q)$ and $write(Q)$ → **CONFLICT**
> - $write(Q)$ and $read(Q)$ → **CONFLICT**
> - $write(Q)$ and $write(Q)$ → **CONFLICT**
>
> If an instruction is just a read, they don't conflict. If at least one instruction is a write, they conflict because the order in which they are executed changes the final value or the value read!

If a concurrent schedule can be transformed into a serial schedule by swapping *non-conflicting* adjacent instructions, we say the schedule is **Conflict Serializable**.

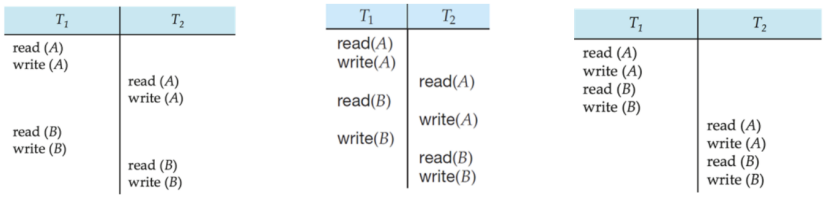


Figure 57.2: Demonstrating how a schedule can be transformed into a serial schedule by swapping non-conflicting instructions.

57.2.1.1 The Precedence Graph

Checking every possible swap is tedious. Instead, we use a **Precedence Graph**:

1. Draw a node for each transaction.
2. Draw a directed edge from T_i to T_j if they have conflicting instructions and T_i executed first.

> **!** The Acyclic Rule
>
> A schedule is conflict serializable **if and only if** its precedence graph contains NO cycles (it is acyclic).

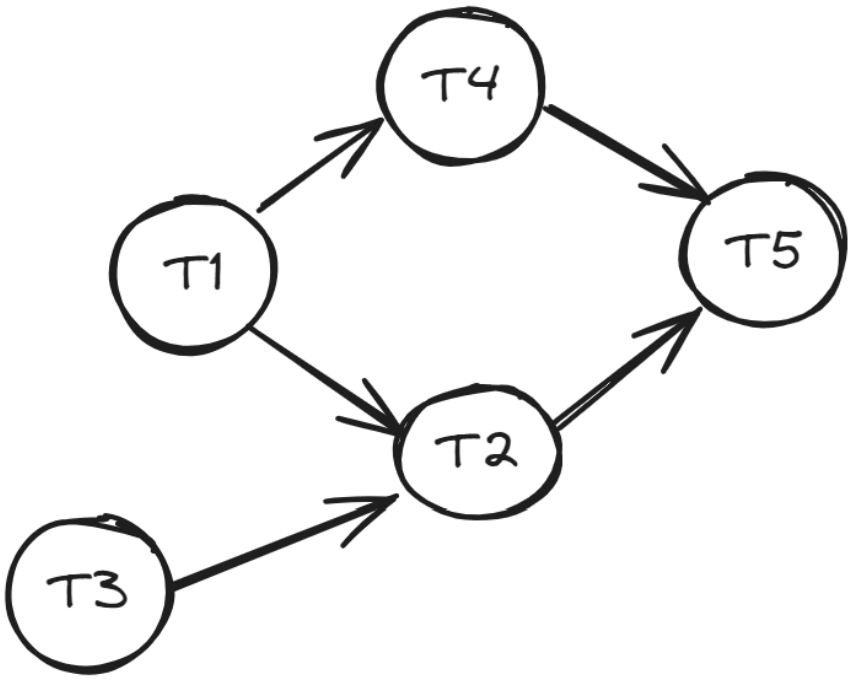


Figure 57.3: A transaction table and its corresponding Precedence Graph. Since there is a cycle (T1 -> T2 -> T1 is possible, though the image shows a complex graph), it would not be conflict serializable.

(Note: If the graph is acyclic, you can use topological sorting to find the exact serial execution order).

57.2.2 2. View Serializability

View serializability is a slightly looser definition than conflict serializability. Two schedules are "view equivalent" if they meet three conditions for every data item:

1. **Initial Read:** If T_i reads the initial value of Q in the first schedule, it must do so in the second.
2. **Write-Read Pair:** If T_j reads a value of Q written by T_i in the first schedule, it must read the value written by that exact same T_i in the second.
3. **Final Write:** The transaction that performs the *last* write to Q must be the same in both schedules.

Every conflict-serializable schedule is view-serializable. However, some view-serializable schedules are *not* conflict-serializable. This usually happens when there are **Blind Writes** (writing to a variable without ever reading it first).

57.3 Recovery and Cascading

If a transaction fails halfway through, we must roll it back to maintain the ACID property of Atomicity. This rollback process is known as **Recovery**. However, rolling back one transaction can cause a domino effect.

57.3.1 Recoverable Schedules

If Transaction B reads data written by Transaction A, then A **must commit before** B commits.

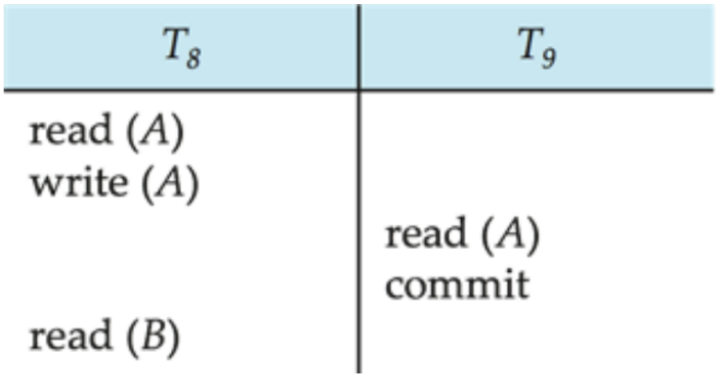


Figure 57.4: A non-recoverable schedule. If T8 aborts after T9 has already committed based on T8's bad data, the database is ruined.

57.3.2 Cascading Rollbacks

Even if a schedule is recoverable, we can run into **Cascading Rollbacks**. If one transaction fails, every other transaction that read its dirty data must *also* be rolled back. This wastes a massive amount of computational work.

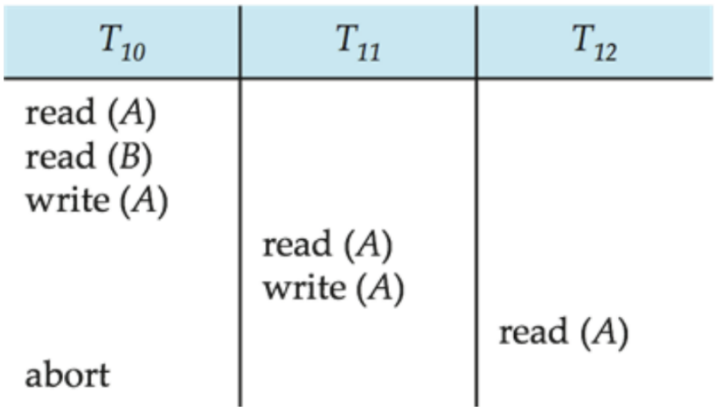


Figure 57.5: A cascading rollback. If T10 fails, T11 and T12 must also be rolled back.

57.3.3 Cascadeless Schedules

To prevent cascading rollbacks, databases enforce **Cascadeless Schedules**. Rule: Transaction B cannot even *read* data written by Transaction A until Transaction A has fully committed. *(Note: Every cascadeless schedule is inherently recoverable).*

57.4 TCL (Transaction Control Language)

How do we actually manage these transactions in SQL? We use TCL commands!

- `COMMIT` - Saves all changes permanently. The transaction is successfully concluded.
- `ROLLBACK` - Undoes all changes made during the current transaction.
- `SAVEPOINT` - Creates a bookmark within a transaction. You can choose to rollback to a specific savepoint instead of rolling back the entire transaction.
- `SET TRANSACTION` - Can be used to set properties, like making the transaction Read-Only.

```
-- An example of using Savepoints
UPDATE Employee SET EName = "Janie" WHERE EID = "E06";
SAVEPOINT SP1;

DELETE FROM Employee WHERE EID = "E02";
SAVEPOINT SP2;

UPDATE Employee SET EName = "Raina" WHERE EID = "E04";

-- Oops, we didn't mean to delete E02! Let's rewind to SP1.
ROLLBACK TO SP1;
```

58 Concurrency and Locking

58.1 The Need for Concurrency Control

We know that a schedule must be both **conflict serializable** and **recoverable** (preferably cascadeless). However, allowing transactions to just run wildly and hoping the schedule turns out serializable is a terrible strategy.

The database needs a **Concurrency Control** mechanism to enforce these rules dynamically as transactions arrive.

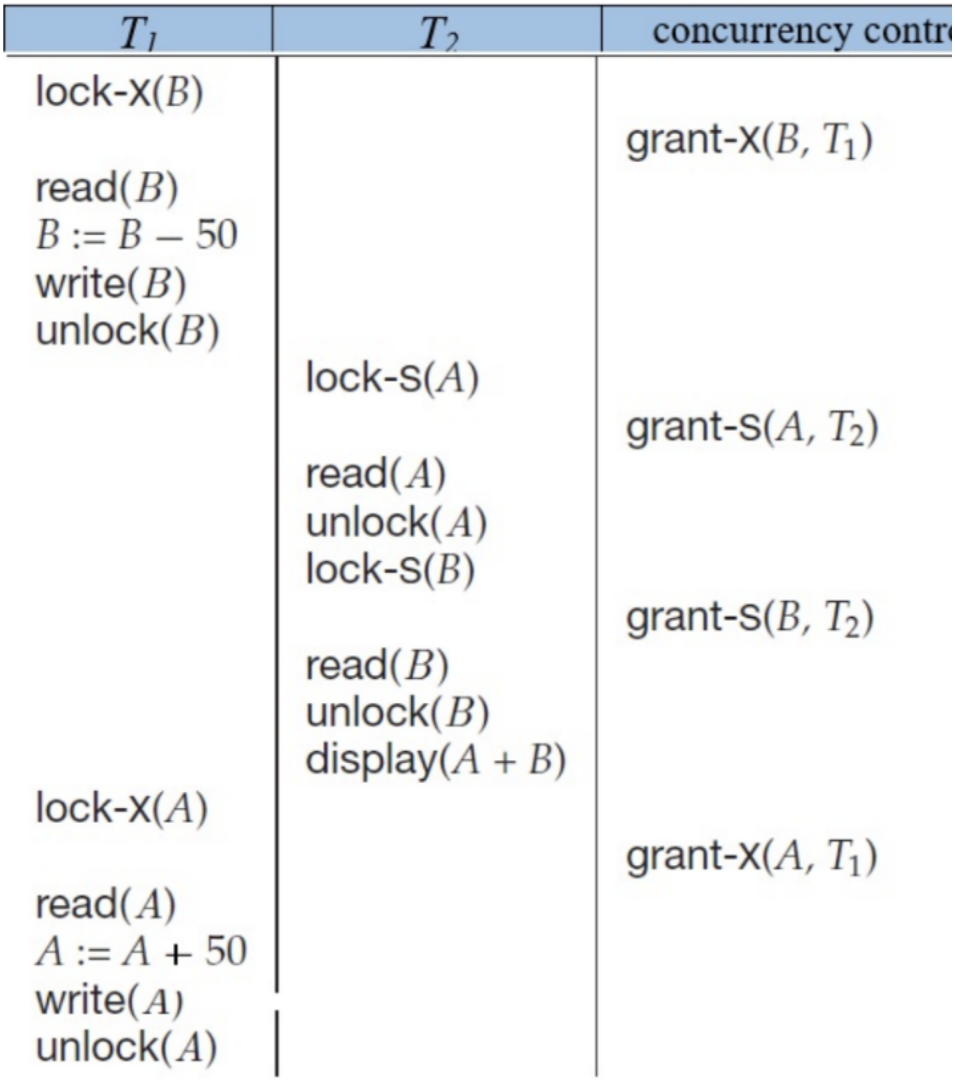


Figure 58.1: A bad concurrent schedule. Because T1 unlocked B too early without proper concurrency control, T2 read an inconsistent state!

There is always a trade-off: The more strict the concurrency control scheme, the safer the data is, but the slower the database runs (because transactions have to wait in line).

58.2 Lock-Based Protocols

The most common way to control concurrency is by using **Locks**. Think of a lock like a bathroom key. If someone has the key, nobody else can enter until they give it back.

Data items can be locked in two modes:

1. **Exclusive (X) mode:** The data item can be both read AND written.
2. **Shared (S) mode:** The data item can ONLY be read.

58.2.1 The Lock-Compatibility Matrix

Before a transaction can lock an item, it must request it from the lock manager. The lock manager checks the compatibility matrix:

State of the lock	Request: Shared	Request: Exclusive
Shared	Yes	No
Exclusive	No	No

- **Sharing is caring:** Any number of transactions can hold Shared locks on the same item at the same time (since reading doesn't change the data).
- **Exclusivity:** If *any* transaction holds an Exclusive lock, no other transaction can hold *any* lock on that item. They must wait.

A transaction must hold a lock on a data item for as long as it is actively accessing it. Once it's done, it releases it using the `unlock(Q)` *instruction.*

58.3 The Two-Phase Locking Protocol (2PL)

Just having locks isn't enough to guarantee serializability. If a transaction locks an item, modifies it, unlocks it, and *then* locks another item, we can still run into consistency issues.

To solve this, databases use the **Two-Phase Locking (2PL)** protocol. Under 2PL, every transaction goes through two distinct phases:

1. **Growing Phase:**
    - A transaction may obtain new locks.
    - It may **not** release any locks.

2. **Shrinking Phase:**
    - A transaction may release locks.
    - It may **not** obtain any new locks.

Once a transaction releases its very first lock, it has entered the Shrinking Phase and can never request another lock again.

58.3.1 Lock Conversions

During these phases, 2PL allows locks to be converted:

- **Growing Phase:** You can upgrade a Shared lock to an Exclusive lock.
- **Shrinking Phase:** You can downgrade an Exclusive lock to a Shared lock.

58.4 Stricter Variants of 2PL

Standard 2PL guarantees serializability, but it does NOT guarantee cascadeless schedules. If a transaction fails during its Shrinking Phase after releasing some locks, other transactions might have already read its dirty data!

To prevent cascading rollbacks, databases use stricter versions of 2PL:

58.4.1 Strict Two-Phase Locking

- Standard 2PL rules apply.
- **Addition:** A transaction must hold all of its **Exclusive** locks until it fully commits or aborts.
- *Why?* This guarantees that no other transaction can read its uncommitted changes.

58.4.2 Rigorous Two-Phase Locking

- Standard 2PL rules apply.
- **Addition:** A transaction must hold **ALL** of its locks (both Shared and Exclusive) until it commits or aborts.
- *Why?* It is even safer and easier to implement, and transactions can be serialized in the exact order they commit.



The Trade-off

As we move from standard 2PL → Strict 2PL → Rigorous 2PL, the system becomes safer and simpler. However, **Concurrency goes down**. Because locks are held longer, other transactions are forced to wait longer, slowing down the overall system performance.

59 Deadlocks and Timestamps

59.1 Understanding Deadlocks

Locking protocols solve the problem of inconsistency, but they introduce a new problem: **Deadlock**.

Imagine Transaction A holds a lock on item X and is waiting for a lock on item Y. Meanwhile, Transaction B holds a lock on item Y and is waiting for a lock on item X.

Neither transaction can ever proceed. They are stuck waiting for each other indefinitely.

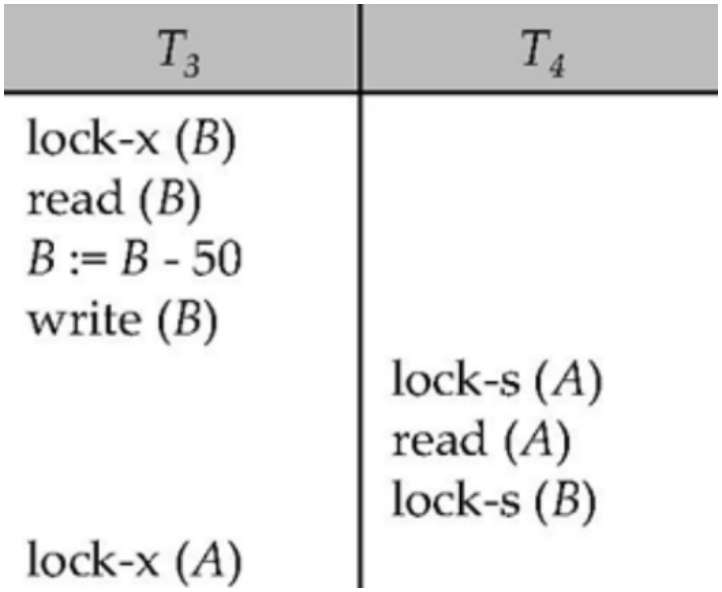


Figure 59.1: A classic deadlock scenario where two transactions are waiting for each other.

When a deadlock occurs, the system has no choice but to intervene. It must **roll back (abort)** one of the transactions to break the cycle and release its locks so the other can proceed.

59.1.1 Starvation

A related issue is **Starvation**. This occurs when a badly designed concurrency control manager repeatedly forces a transaction to wait forever, even without a deadlock. For example, if Transaction A wants an Exclusive lock, but a continuous stream of new transactions keep requesting and getting Shared locks on that item, Transaction A will starve. Alternatively, a transaction might be repeatedly chosen as the "victim" to be rolled back during deadlocks, never getting a chance to finish.

59.2 Deadlock Prevention

Rather than letting deadlocks happen and fixing them later, databases can use schemes to prevent them from ever occurring. These schemes often use **Transaction Timestamps**.

A Timestamp is a unique identifier assigned by the database that represents the exact moment a transaction started. An older transaction has a smaller timestamp, while a younger transaction has a larger timestamp.

59.2.1 1. The Wait-Die Scheme (Non-Preemptive)

Rule: Older transactions wait for younger ones. Younger transactions die.

If Transaction A requests a lock held by Transaction B:

- If A is **older** than B $\rightarrow$ A is allowed to **wait**.
- If A is **younger** than B $\rightarrow$ A is killed (**dies**) and rolled back.

59.2.2 2. The Wound-Wait Scheme (Preemptive)

Rule: Younger transactions wait for older ones. Older transactions wound (kill) younger ones.

If Transaction A requests a lock held by Transaction B:

- If A is **older** than B $\rightarrow$ A forces B to be killed (**wounds** it). B is rolled back and restarted later.
- If A is **younger** than B $\rightarrow$ A is allowed to **wait**.

59.3 Deadlock Detection

If a database doesn't use prevention schemes, it must periodically check for deadlocks using a **Wait-For Graph**.

- **Vertices (Nodes):** Represent active transactions.
- **Edges:** A directed edge from T_i to T_j means that T_i is waiting for T_j to release a lock.

> **!** The Cycle Rule
>
> The system is in a deadlock state **if and only if** the wait-for graph has a cycle.

If a cycle is detected, the database selects a "victim" transaction and rolls it back.

59.4 Timestamp-Based Protocols

What if we didn't use locks at all? Another way to handle concurrency is by strictly enforcing execution based on timestamps.

Every data item Q in the database maintains two timestamps:

- **W-timestamp(Q):** The largest timestamp of any transaction that successfully *wrote* to Q.
- **R-timestamp(Q):** The largest timestamp of any transaction that successfully *read* Q.

59.4.1 Read and Write Rules

Suppose Transaction T_i wants to **read(Q)**:

- If $TS(T_i) < W\text{-timestamp}(Q)$: This means a *younger* transaction has already overwritten the data! T_i is trying to read the past. The read is rejected, and T_i is rolled back.
- If $TS(T_i) \geq W\text{-timestamp}(Q)$: The read is allowed! We update R-timestamp(Q) if T_i's timestamp is larger.

Suppose Transaction T_i wants to **write(Q)**:

- If $TS(T_i) < R\text{-timestamp}(Q)$: A younger transaction has already read the data assuming T_i's write never happened. The write is rejected, and T_i is rolled back.

- If $TS(T_i) < W\text{-timestamp}(Q)$: A younger transaction has already overwritten the data. The write is rejected, and T_i is rolled back.
- Otherwise, the write is allowed, and $W\text{-timestamp}(Q)$ is updated.

59.4.2 Correctness and Drawbacks

- **Pros:** It guarantees serializability and inherently prevents deadlocks (because no transaction ever waits for a lock—they just execute or get rolled back).
- **Cons:** It does NOT guarantee cascadeless or recoverable schedules, so transaction failures can cause significant cascading rollbacks.

60 (Optional) Real-World Concurrency

60.1 Theory vs. Practice

Throughout this week, we've explored strict theoretical protocols like Two-Phase Locking (2PL) and Timestamp-based protocols. However, if you spin up a real, modern database like **PostgreSQL** or **MySQL (InnoDB)**, you'll find that they don't *strictly* use standard 2PL for everything.

Why? Because standard 2PL forces reads to block writes, and writes to block reads. In a massive web application with thousands of concurrent users, this level of locking would cause unacceptable performance bottlenecks.

Modern databases solve this using a hybrid approach called **MVCC**.

60.2 Multi-Version Concurrency Control (MVCC)

MVCC is the secret sauce behind almost all modern enterprise relational databases.

The core philosophy of MVCC is simple: **"Readers do not block Writers, and Writers do not block Readers."**

60.2.1 How does it work?

Instead of locking a row when it's being updated, the database simply creates a *new version* of that row.

1. **Transaction A** wants to update a row (e.g., change Alice's balance from \$100 to \$50). It writes a brand new version of the row to the disk but doesn't delete the old one yet.
2. **Transaction B** comes along and wants to read Alice's balance.
3. Because Transaction A hasn't committed yet, Transaction B ignores the new version and reads the *old version* (\$100). **No locks required!**
4. Once Transaction A commits, any *future* transactions will see the new version (\$50).

Under the hood, this uses the **Timestamp Protocols** we learned about! Every transaction gets an ID (a timestamp), and every row version is tagged with the transaction IDs that created and deleted it. The database uses these tags to mathematically determine which version of a row a specific transaction is allowed to "see".

60.3 Isolation Levels in Practice

Because perfect serializability is expensive, SQL standards allow developers to choose their desired **Isolation Level** for a transaction. This allows you to trade safety for speed depending on your specific use case.

From most dangerous (fastest) to safest (slowest):

1. **Read Uncommitted:** A transaction can read uncommitted "dirty" data from other transactions. (Extremely fast, but very dangerous).
2. **Read Committed:** (Default in PostgreSQL) A transaction can only read data that has been fully committed. However, if it reads the same row twice, the value might change if another transaction committed an update in between.
3. **Repeatable Read:** (Default in MySQL) Guarantees that if you read a row twice in the same transaction, you will see the exact same data both times, even if another transaction updated it.
4. **Serializable:** The absolute strictest level. The database guarantees the result is mathematically identical to serial execution. If it detects a conflict that breaks serializability, it simply aborts one of the transactions.

60.3.1 The Trade-off

Most companies do not run their databases in `Serializable` mode because the performance penalty and frequency of deadlocks/aborts is too high. They run in `Read Committed` or `Repeatable Read`, accepting minor theoretical inconsistencies in exchange for massive concurrency gains.

Part XI

Week-11

61 Backup and Recovery

61.1 What is Backup and Recovery?

Imagine your database crashes due to a power failure. Without a backup, everything is gone. A **backup** is a copy of your database — data files, control files, logs — that lets you restore the system when something goes wrong.

Recovery is the process of using that backup to bring the database back to its last consistent state.

61.2 Types of Data to Back Up

- **Business Data** — includes personal information of clients, employees, contractors etc. along with details about places, things, events and rules related to the business.
- **System Data** — includes specific environment/configuration of the system used for specialised development purposes log files, software dependency data, disk images.
- **Media Files** — files like photographs, videos, sounds, graphics etc. need backing up. Media files are typically much larger in size.

61.3 Backup Strategies

61.3.1 Full Backup

A full backup copies *everything* — tables, indexes, views, procedures. Think of it as a complete snapshot of the database at a point in time. It must be done **at least once** before any other strategy can be used.

Advantages	Disadvantages
Simple — recovery needs just one backup set	Slowest to perform
No dependency between backups	Longest system downtime
Loss of one backup doesn't affect others	Uses the most storage

---

61.3.2 Incremental Backup

Instead of copying everything, an incremental backup only copies what has **changed since the last backup** (full or incremental).

Advantages	Disadvantages
Smallest backup size	Recovery needs full + every incremental in sequence
Minimal downtime	Missing even one incremental breaks recovery
Most cost-efficient	Most complex to manage

61.3.3 Differential Backup

A differential backup copies everything that has **changed since the last full backup** — regardless of any incrementals in between. Recovery only ever needs two things: the last full backup and the most recent differential.

The longer you wait between differentials, the larger they grow — eventually approaching full backup size.

Advantages	Disadvantages
Fewer backup sets needed for recovery	Larger than incremental backups
Great when full backups are infrequent	Grows over time if not managed

---

61.3.4 Comparing the Three

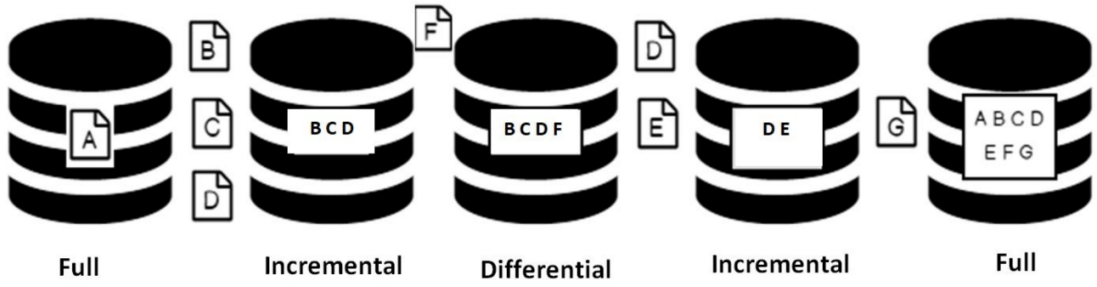


Figure: Backup Types

	Full	Incremental	Differential
What it copies	Everything	Changes since last backup	Changes since last full
Backup size	Largest	Smallest	Medium (grows over time)
Recovery sets needed	1	Full + all incrementals	Full + latest differential
Recovery complexity	Simplest	Most complex	Moderate

A smart schedule: **full backup monthly + differential weekly + incremental daily**.

61.4 Hot Backup

All the strategies above are **cold backups** — the database pauses during the backup. But some systems can never go offline: financial trading platforms, real-time sensor systems, satellite feeds.

Hot backup keeps the database fully live while the backup runs concurrently.

Advantages	Disadvantages
Database stays available to users	Not feasible for very large, monolithic datasets

Advantages	Disadvantages
Point-in-time recovery is easier	Any error mid-backup can terminate the entire process
Efficient for dynamic, modular data	High setup and maintenance cost

In practice: hot backup is used for **transaction log backups**. Data backups (incremental/differential) are done cold. A **transaction log** records every database modification in sequence. Even if a data backup is slightly inconsistent (taken while the database was live), replaying the transaction log on top of it restores full consistency.

62 Log-Based Recovery

62.1 How Data Moves: The Access Model

Data lives on **disk** (physical blocks) and is loaded into **memory** (buffer blocks) when needed.

- `input(B)` — load block B from disk into memory buffer.
- `output(B)` — flush buffer block B back to disk.

Inside a transaction:

- `read(X)` — copy data item X from buffer into the transaction's local variable.
- `write(X)` — copy the local variable back into the buffer block.

62.2 Log-Based Recovery

The key idea: before touching the database, every change is first recorded in a **log** on stable storage. If a crash happens, the log tells us exactly what to undo or redo. A log is kept on stable storage. The log is a sequence of log records, which maintains information about update activities on the database.

Log record types:

Record	Written when
`< Ti start >`	Transaction begins
`< Ti, X, V1, V2 >`	Before writing X — V1 is the old value, V2 is the new value
`< Ti commit >`	Transaction completes successfully
`< Ti abort >`	Transaction is rolled back

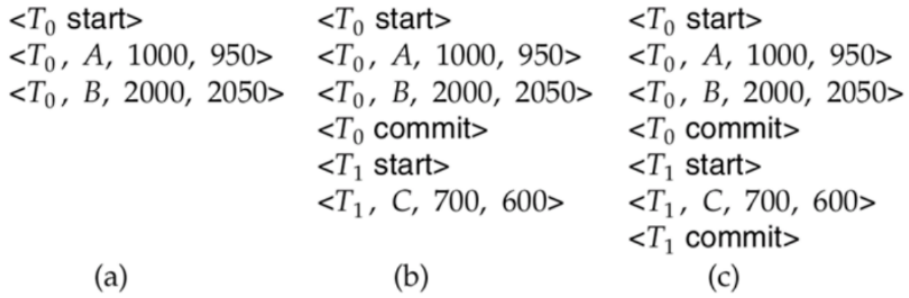
A transaction is considered **committed** the moment its commit log record reaches stable storage — even if the actual data changes are still sitting in the buffer.

62.3 Database Modification Schemes

Scheme	When updates hit disk
Immediate modification	Anytime — before or after commit
Deferred modification	Only at commit time

We focus on **immediate modification** — it requires both undo and redo during recovery, but it is the more commonly used approach.

62.3.0.1 Example



Scenario	Log state at crash	Recovery action
(a)	T0 started but neither committed nor aborted	**Undo T0** — restore B to 2000, then A to 1000
(b)	T0 committed; T1 started but not committed	**Redo T0** (set A=950, B=2050); **Undo T1** (restore C to 700)
(c)	Both T0 and T1 committed	**Redo T0** then **Redo T1** (set A=950, B=2050, C=600)

62.4 Undo and Redo

When a crash happens, the recovery system needs to either reverse incomplete work or replay committed work that didn't make it to disk.

Operation	What it does	Direction	When used
`undo(Ti)`	Restores all writes to old values (V1)	Scans log **backwards**	Ti started but never committed
`redo(Ti)`	Reapplies all writes to new values (V2)	Scans log **forwards**	Ti committed but data may not be on disk

During undo, each restored value is recorded as a **Compensation Log Record (CLR)**: `< Ti, X, V1 >`. When the undo is complete, `< Ti abort >` is written.

62.5 Deciding What to Do After a Crash

Log contains	Action
`< Ti start >` but no commit or abort	**Undo** Ti
`< Ti start >` and `< Ti commit >` or `< Ti abort >`	**Redo** Ti

62.6 Checkpoints

Without checkpoints, recovery would replay the entire log from the very beginning — extremely slow for a database that has been running for a long time. A **checkpoint** flushes all buffered writes to disk and records which transactions are currently active. Recovery only needs to look at the log from the checkpoint onwards.

Steps during checkpointing:

1. Output all in-memory log records to stable storage.
2. Flush all modified buffer blocks to disk.
3. Write a `< checkpoint L >` record, where L is the list of active transactions.

After a crash, for each transaction relative to the last checkpoint:

Transaction state	Action
Committed **before** checkpoint	Ignore — already safely on disk
Committed **after** checkpoint	Redo
Still active at time of crash	Undo

---

63 RAID: Redundant Array of Independent Disks

63.1 What is RAID?

A single disk has two problems: it's slow, and it can fail. RAID solves both by combining multiple disks into what looks like one — spreading data across them for speed and duplicating it for safety. **RAID (Redundant Array of Independent Disks)** is a disk organization technique that manages multiple disks and presents them as a single logical disk. It achieves two things simultaneously:

- **Speed** — by using multiple disks in parallel, it delivers high capacity and high throughput.
- **Reliability** — by storing data redundantly across disks, it ensures data can be recovered even if one disk fails.

63.2 Three Core Techniques

63.2.1 Mirroring

Every disk has an exact duplicate — the logical disk consists of two physical disks.

- Every write is carried out on **both** disks simultaneously.
- Reads can be served from **either** disk.
- If one disk in a pair fails, data is still available on the other.
- Data loss only occurs if both disks in a pair fail before one is repaired — extremely unlikely.

63.2.2 Striping

Data is split across multiple disks so reads and writes happen in parallel.

Bit-level Striping

- Bits of each byte are split across multiple disks (bit i goes to disk i).
- Each access reads data at up to $8\times$ the rate of a single disk.
- Seek and access time is worse than a single disk — rarely used today.

Byte-level Striping

- Each file is split into 1-byte chunks distributed across disks.
- The ith byte goes to disk $((i-1) \mod n)+1$.

Block-level Striping

- Block i of a file goes to disk $(i \mod n)+1$.
- Requests for different blocks can run in **parallel** if they reside on different disks.
- A long sequential read can utilise all disks at once.

63.2.3 Parity

Parity provides fault tolerance by storing extra computed information alongside the data. If a disk fails, the lost data can be **reconstructed** from the remaining disks and the parity — without needing a full duplicate copy.

Parity is computed using **XOR**: for any set of bits, XOR of all bits including the parity bit equals zero. So if one disk's bits are lost, XOR of the remaining disks recovers them.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 63.1: XOR Gate

Bit-interleaved Parity

- One parity bit covers the corresponding bits across all data disks.
- Written to a dedicated parity disk on every write.
- On disk failure: XOR all remaining bits (including parity) to reconstruct the lost bit.

Block-interleaved Parity

- Uses block-level striping with a dedicated parity **block** (not just a bit) for every set of corresponding blocks across n disks.
- On disk failure: XOR the corresponding blocks from all remaining disks to reconstruct the lost block.

63.3 Standard RAID Levels

63.3.1 RAID 0 — Block Striping

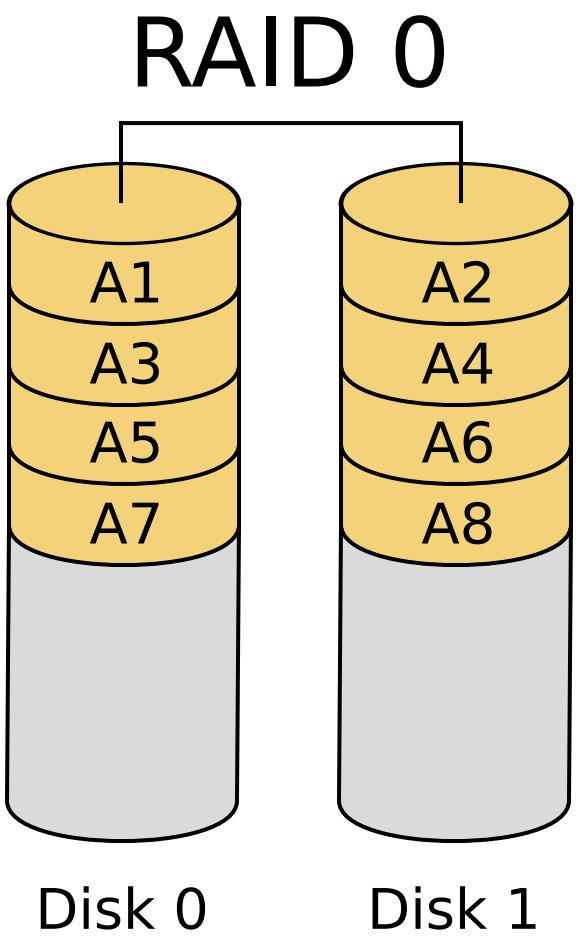


Figure 63.2: RAID 0

- Data is striped across disks with no redundancy whatsoever.
- Best write performance of all RAID levels — no parity to update.
- 100% storage efficiency.
- No fault tolerance — one disk failure loses everything.

63.3.2 RAID 1 — Mirroring

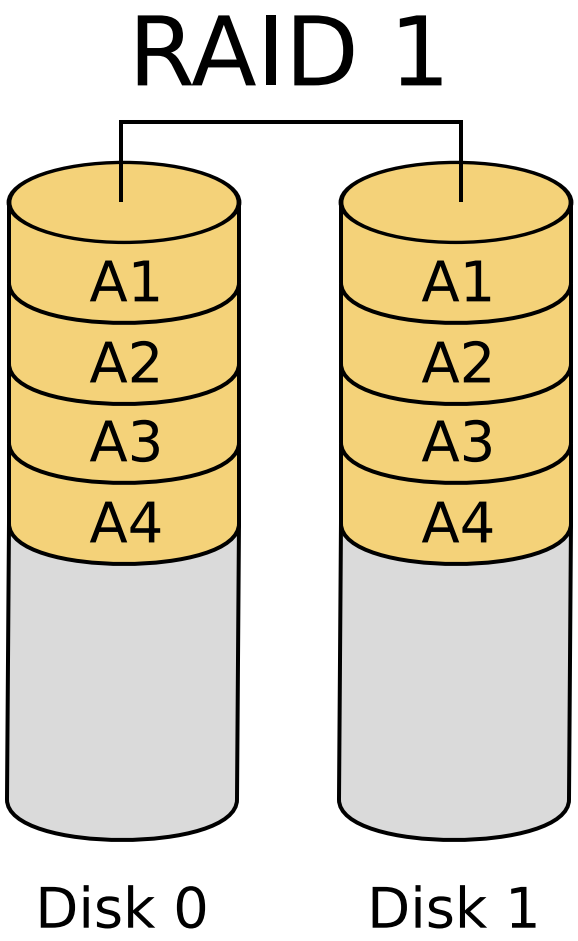


Figure 63.3: RAID 1

- Every disk has an exact copy on a second disk. Every write of a disk block involves a write on both disks.
- Excellent fault tolerance — survives one disk failure.
- Reads can be served from either disk in parallel.
- 50% storage efficiency — half your disks are copies.
- Most expensive solution.

63.3.3 RAID 2 — Parity

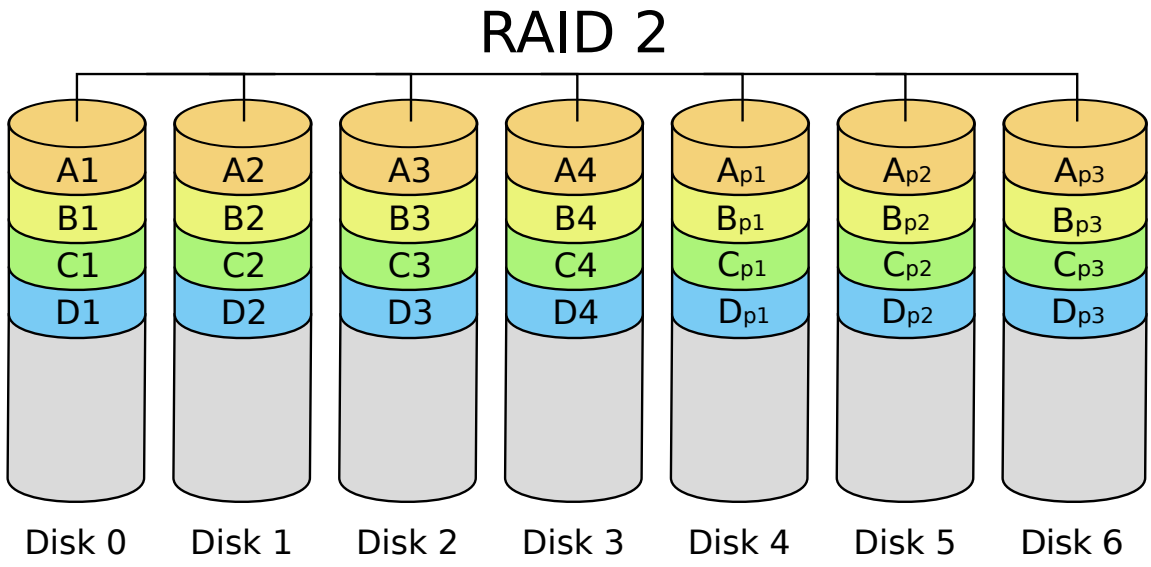


Figure 63.4: RAID 2

- Uses dedicated bit for parity and the striping is a single bit, uses Hamming codes for error correction.
- Can correct one-bit errors and detect two-bit errors.
- Requires multiple dedicated check disks.
- Not used in practice — subsumed by RAID 3.

63.3.4 RAID 3 — Byte Striping + Parity

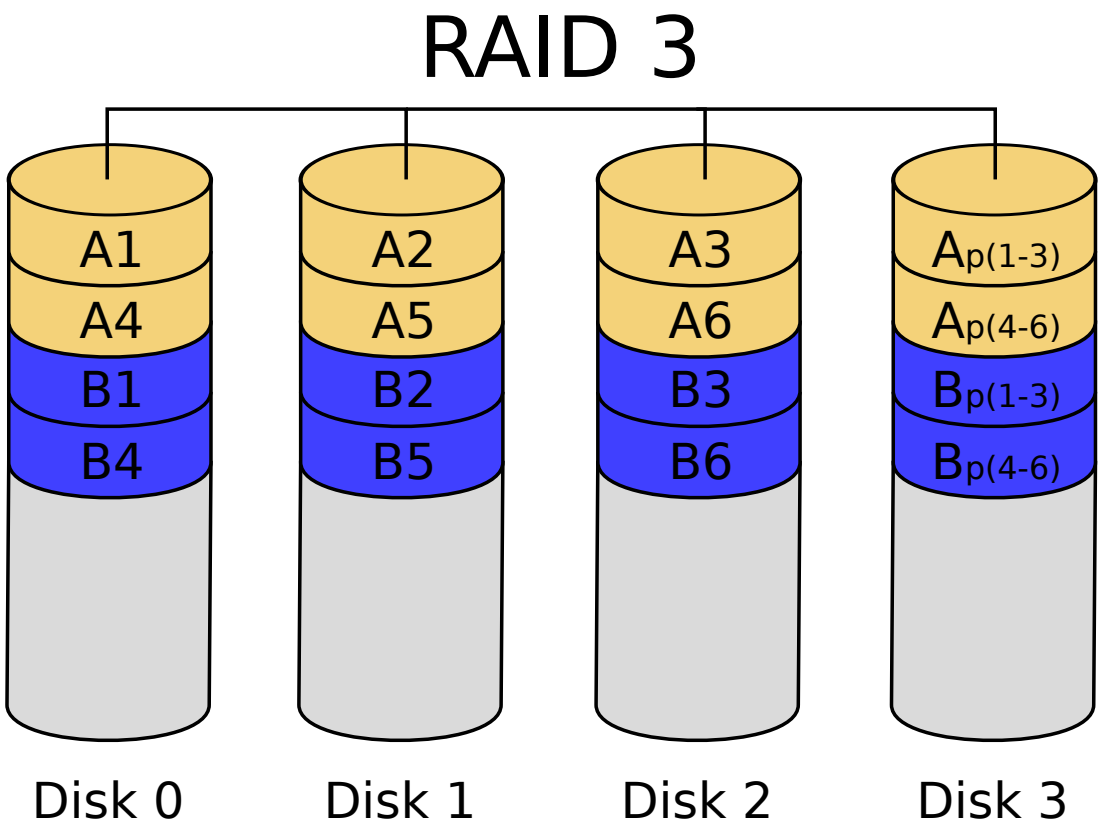


Figure 63.5: RAID 3

- Strips at the byte level with a single dedicated parity disk.
- High data transfer rate — data accessed in parallel across all disks.
- Cannot service multiple requests simultaneously — every request touches every disk.
- Reliability overhead for RAID 3 is a single disk, the lowest over-head possible.

63.3.5 RAID 4 — Block Striping + Parity

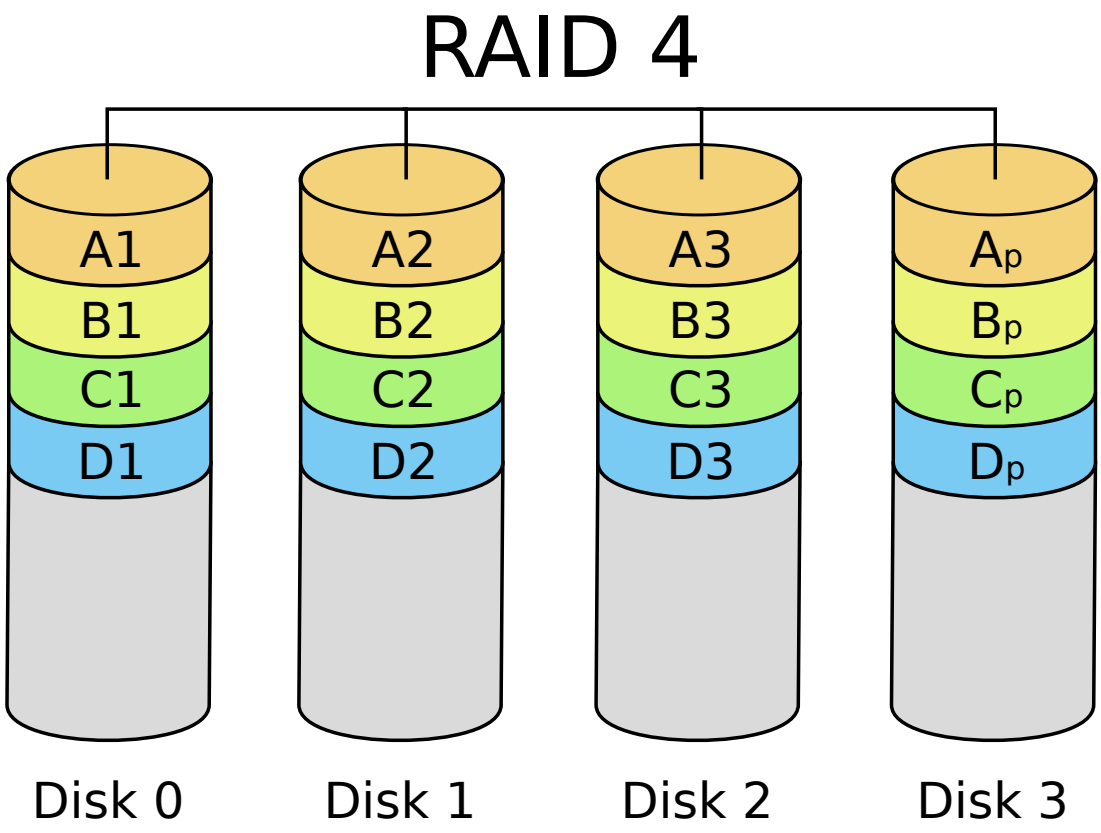


Figure 63.6: RAID 4

- Strips at the block level with a single dedicated parity disk.
- Read requests can be served by a single disk — better than RAID 3 for reads.
- Can recover from one disk failure using XOR.
- Write performance is poor — every write hits the single parity disk, creating a bottleneck.

63.3.6 RAID 5 — Distributed Parity

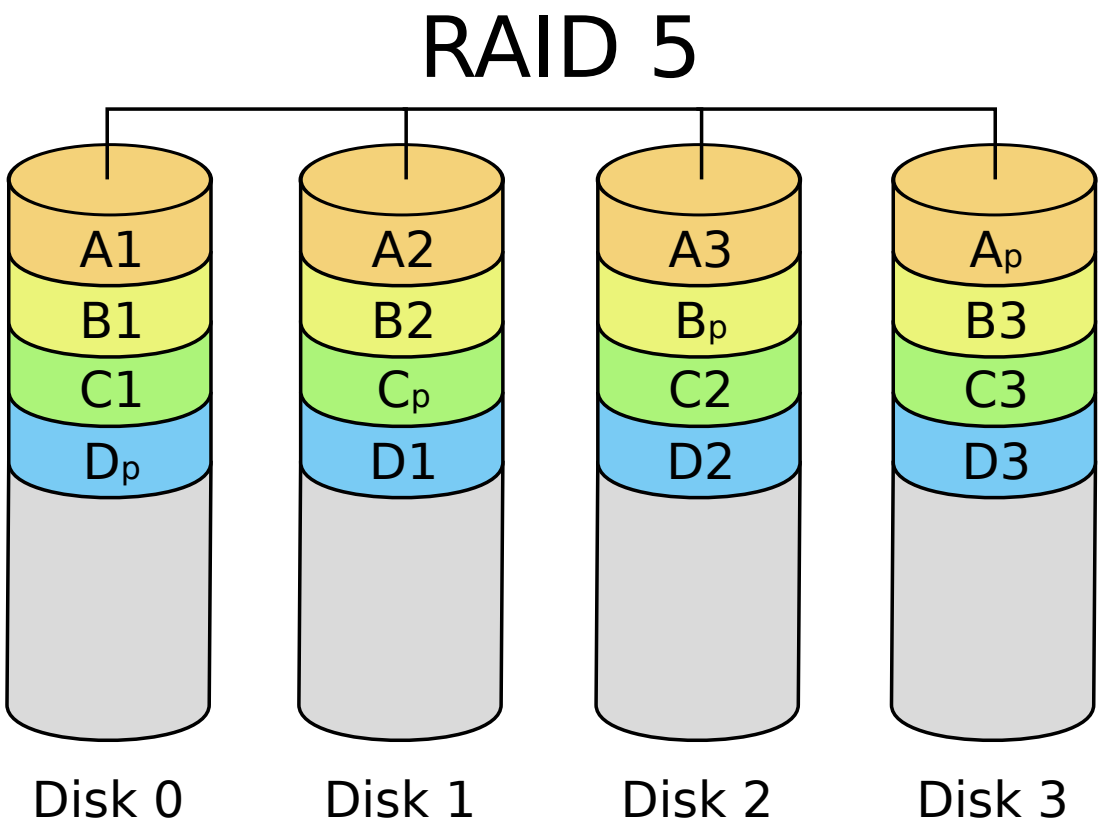


Figure 63.7: RAID 5

- Like RAID 4 but parity blocks are spread across all disks instead of sitting on one dedicated disk.
- Eliminates the parity bottleneck — multiple writes can happen in parallel.
- All disks participate in reads — higher read parallelism.
- Allows recovery of only 1 disk failure.

63.3.7 RAID 6 — Dual Parity

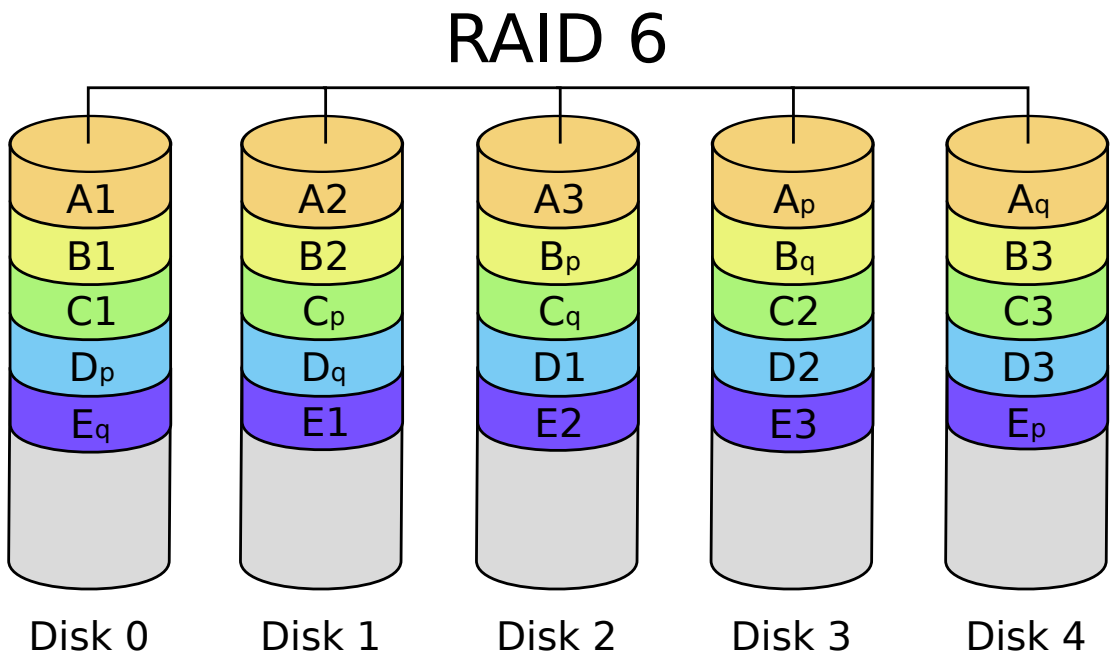


Figure 63.8: RAID 6

- Extends RAID 5 with a second parity block, using Reed-Solomon codes.
- Survives **two simultaneous disk failures** — safe even during a rebuild after one failure.
- Write performance worse than RAID 5 — two parity blocks to update.

63.3.8 RAID 01 — Mirror of Stripes

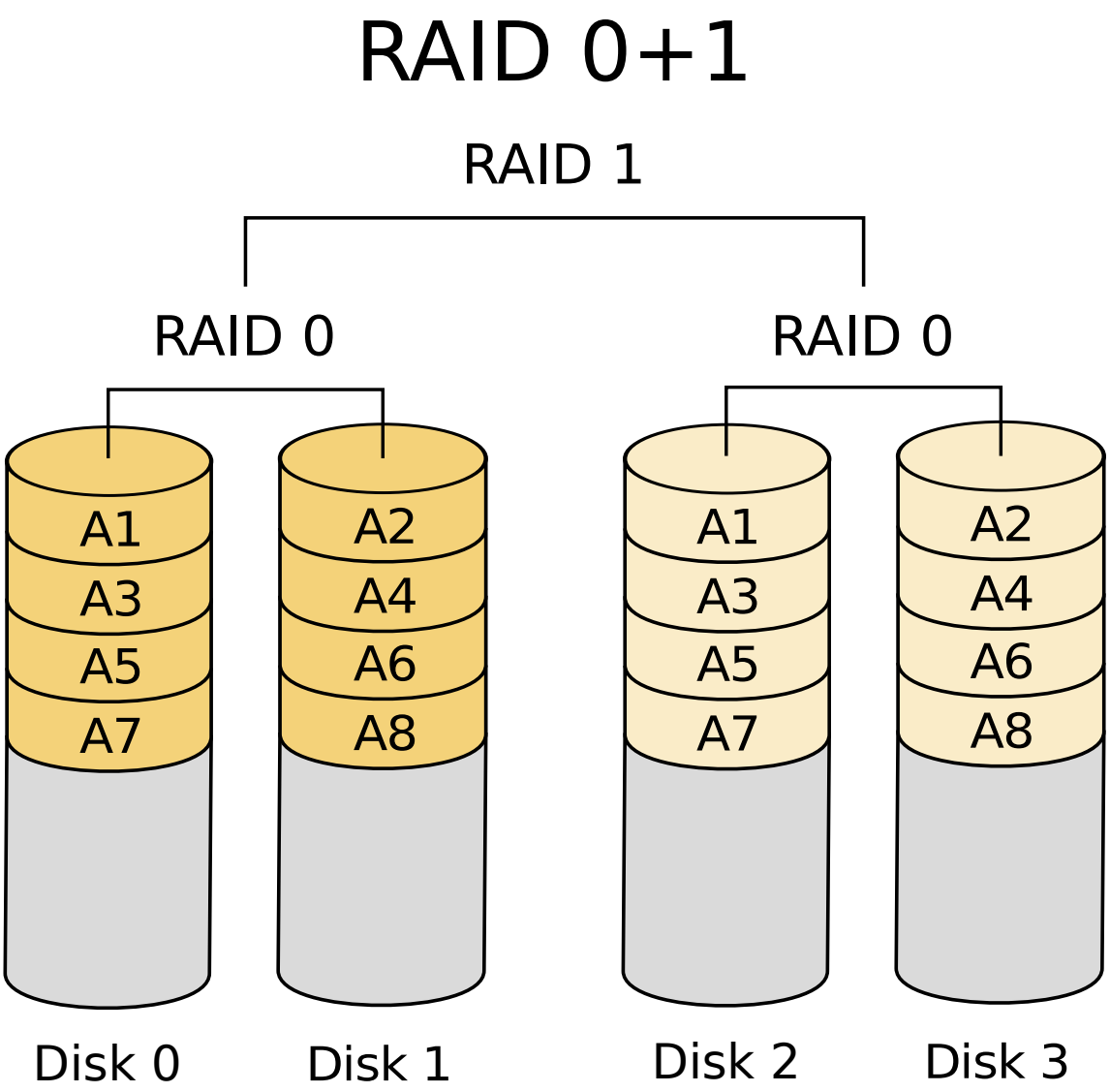


Figure 63.9: RAID 01

- Stripe sets are built first (RAID 0), then each set is mirrored (RAID 1).
- If one disk in a stripe fails, the entire stripe set is lost — falls back to the mirror.
- Less resilient than RAID 10.
- Requires at least 4 disks. Storage efficiency: 50%.

63.3.9 RAID 10 — Stripe of Mirrors

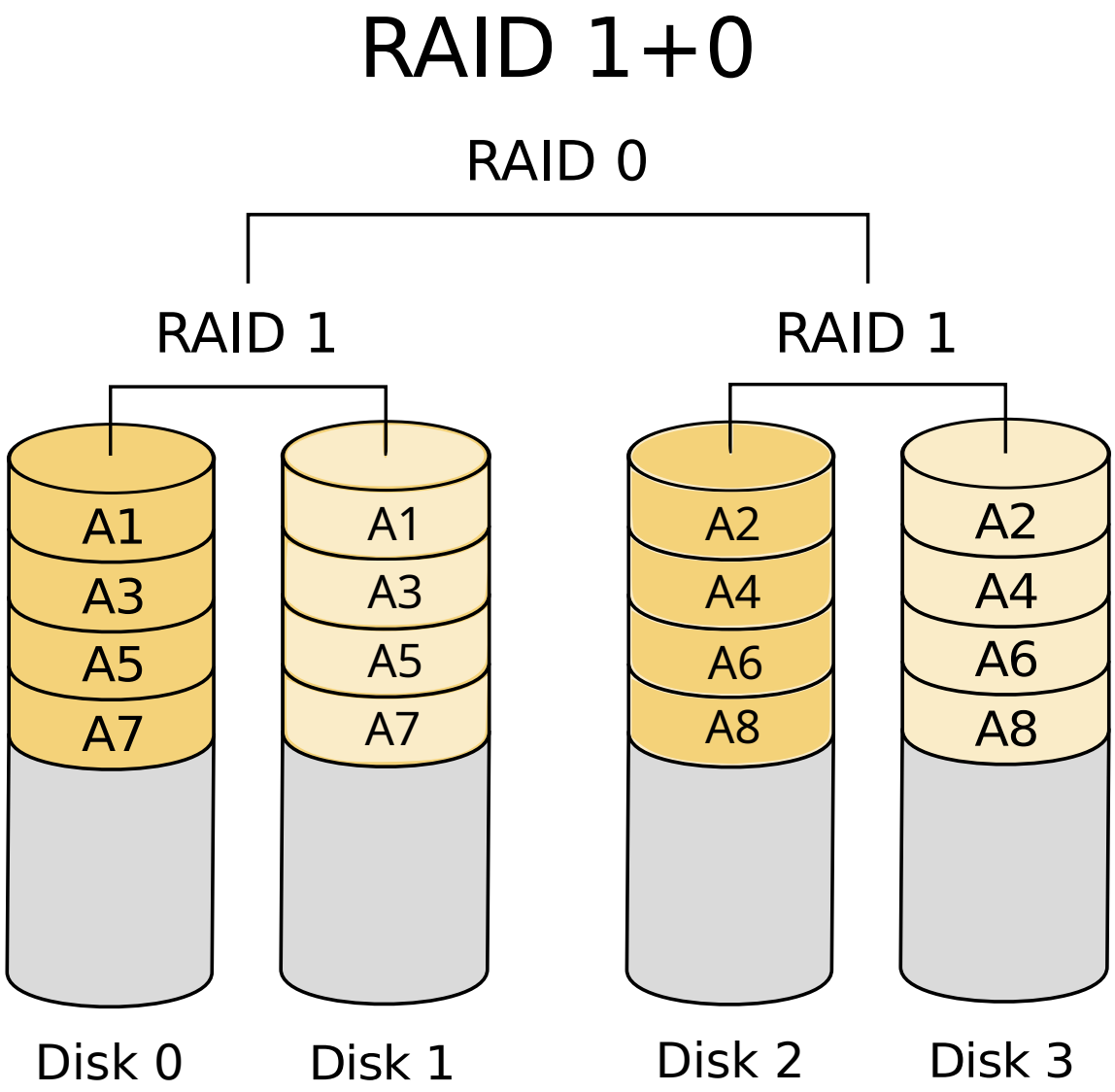


Figure 63.10: RAID 10

- Disks are mirrored first (RAID 1), then striped across the mirror pairs (RAID 0).
- One disk in each mirror pair can fail independently — far more resilient than RAID 01.

- Best throughput and latency of all fault-tolerant RAID levels.

Part XII

Week-12

64 Query Processing and Cost

64.1 Overview of Query Processing

When you submit a query to a database, it doesn't just execute it directly. It goes through a series of logical steps to ensure the query is valid and to find the most efficient way to execute it.

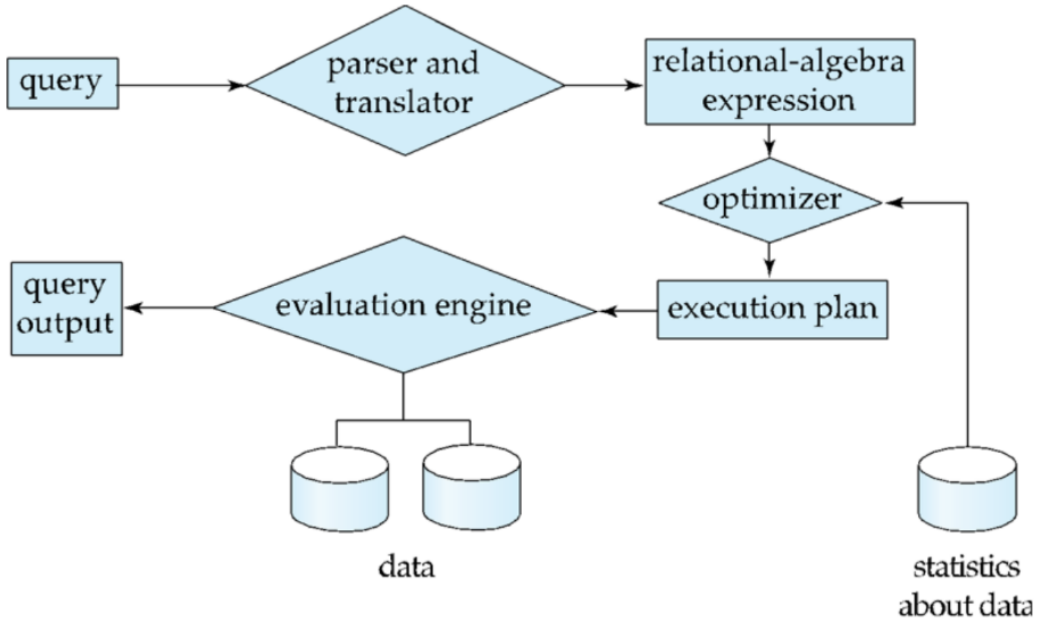


Figure 64.1: The phases of query processing.

The process can be broken down into three main phases:

64.1.1 1. Parser and Translator

- **What it does:** The database receives the raw SQL query. It checks the syntax to ensure it's valid SQL, and verifies that the referenced tables and columns exist in the database schema.

- **Output:** It translates the query into an internal format, typically a relational algebra expression.

64.1.2 2. Optimization

- **What it does:** This is the most crucial step for performance. A single SQL query can often be executed in multiple ways (e.g., different join orders, using an index vs. scanning a table).
- **The Optimizer's Job:** It analyzes these different strategies (execution plans), estimates the cost of each using available database statistics, and selects the one with the lowest overall cost.

64.1.3 3. Evaluation

- **What it does:** The query-execution engine takes the optimized query-evaluation plan and actually runs it against the data.
- **Output:** It returns the final result set (the answers) to the user.

64.2 Measures of Query Cost

How does the optimizer determine the "cost" of a query plan?

The true cost is generally measured as the **total elapsed time** required to answer the query. This involves several factors:

- Disk accesses (reading/writing data from/to storage)
- CPU time (processing the data)
- Network communication (if data is distributed)

However, because fetching data from disk is typically the slowest operation by a huge margin, we simplify the cost measure to focus primarily on disk I/O.

We measure cost using two main metrics:

1. **Number of block transfers (b):** The number of data blocks that must be read from or written to the disk.
2. **Number of seeks (S):** The number of times the disk head must move to a new location to read a block.

64.2.1 The Cost Formula

We can estimate the time cost with this simple formula:

$$\text{Cost} = (b \times t_T) + (S \times t_S)$$

Where:

- b = Total number of block transfers
- t_T = Time to transfer one block
- S = Total number of seeks
- t_S = Time for one seek

65 Join Algorithms

65.1 Understanding Join Costs

Joining tables is one of the most common and resource-intensive operations in a database. Therefore, optimizing how a join is executed is critical.

Let's look at three fundamental algorithms for performing joins and how their costs are calculated.

Terminology: When joining two relations (tables), r and s:

- r = The outer relation
- s = The inner relation
- n_r = Number of tuples (rows) in the outer relation r
- b_r = Number of blocks occupied by the outer relation r
- n_s = Number of tuples in the inner relation s
- b_s = Number of blocks occupied by the inner relation s

65.2 1. Nested Loop Join

This is the simplest and most naive join algorithm. It uses two nested `for` loops.

Analogy: Imagine having two physical phone books. For every single person in Book A, you flip through the entire Book B page-by-page to find a matching name. Very tedious!

How it works: For every single tuple in the outer relation r, it scans through every single tuple in the inner relation s to find a match.

```
for each tuple tr in r do
    for each tuple ts in s do
        if tr and ts match the join condition then
            add (tr, ts) to the result
```

Cost:

- **Block Transfers:** $n_r \times b_s + b_r$ (For each of the n_r tuples, it reads all b_s blocks of the inner relation, plus the initial b_r blocks of the outer relation).
- **Seeks:** $n_r + b_r$

65.3 2. Block Nested Loop Join

This algorithm is a major improvement over the standard nested loop join. Instead of looping tuple-by-tuple, it processes the data block-by-block.

Analogy: Instead of looking up one person at a time, you tear out a whole page of names from Book A, memorize them, and then flip through Book B looking for matches for *anyone* on your memorized page.

How it works: For every *block* in the outer relation r, it scans every *block* in the inner relation s. It then joins the tuples within those loaded blocks in memory. This drastically reduces the number of times the inner table has to be read from disk.

```
for each block Br in r do
    for each block Bs in s do
        for each tuple tr in Br do
            for each tuple ts in Bs do
                if tr and ts match then
                    add (tr, ts) to the result
```

Cost:

- **Block Transfers:** $b_r \times b_s + b_r$ (For each of the b_r blocks, it reads all b_s blocks of the inner relation, plus the initial b_r blocks).
- **Seeks:** $2 \times b_r$

65.4 3. Indexed Nested-Loop Join

This algorithm makes use of existing indexes to completely skip scanning the inner relation.

Analogy: Instead of flipping through Book B blindly, you use the alphabetized index at the back of Book B to jump exactly to the page containing the name you're looking for from Book A.

How it works: If the inner relation s has an index on the join attribute, the database can use it. For each tuple t_r in the outer relation r, it simply performs an index lookup on s to instantly find the matching tuples.

Cost:

- **Cost of the join:** $b_r \times (t_T + t_S) + n_r \times c$
  - Where c is the cost of traversing the index and fetching all matching s tuples for one tuple in r.
- **Best Practice:** If both relations have an index, the database optimizer will choose the relation with *fewer tuples* as the outer relation (r) to minimize the number of index lookups.

65.5 Example Scenario

Let's compare these algorithms with some numbers.

Relation	student (Outer)	takes (Inner)
Number of tuples (n)	5,000	10,000
Number of blocks (b)	100	400

65.5.1 Cost using Nested Loop Join

Using `student` as the outer relation (r):

- **Block Transfers:** $5000 \times 400 + 100 = 2,000,100$
- **Seeks:** $5000 + 100 = 5,100$

65.5.2 Cost using Block Nested Loop Join

Using `student` as the outer relation (r):

- **Block Transfers:** $100 \times 400 + 100 = 40,100$
- **Seeks:** $2 \times 100 = 200$

65.5.3 Cost using Indexed Nested-Loop Join

Imagine `takes` has a B+-tree index on the attribute ID. The height of the tree is 4, plus one access to find the actual data, so $c \approx 5$ accesses per lookup. Using `student` as the outer relation (r):

- **Cost (in block transfers and seeks):** $100 + (5000 \times 5) = 25,100$

Conclusion:

- The **Block Nested Loop Join** (40,100 transfers) is drastically more efficient than the basic nested loop (2 million transfers) because it reads data in chunks.
- The **Indexed Nested-Loop Join** (25,100 accesses) is even faster because it completely avoids scanning the `takes` table, jumping directly to the correct records using the index!

66 Query Optimization and Equivalence

66.1 Query Optimization Example

A single query can be executed in multiple ways depending on the order of operations. The database optimizer analyzes these options to choose the most efficient path.

Consider a database with these schemas:

- **course** (course_id, title, dept_name, credits)
- **instructor** (ID, name, dept_name, salary)
- **teaches** (ID, course_id, sec_id, semester, year)

Goal: Select the course instructor name and title who belongs to the department 'Music'.

Here are two possible evaluation plans for the exact same query:

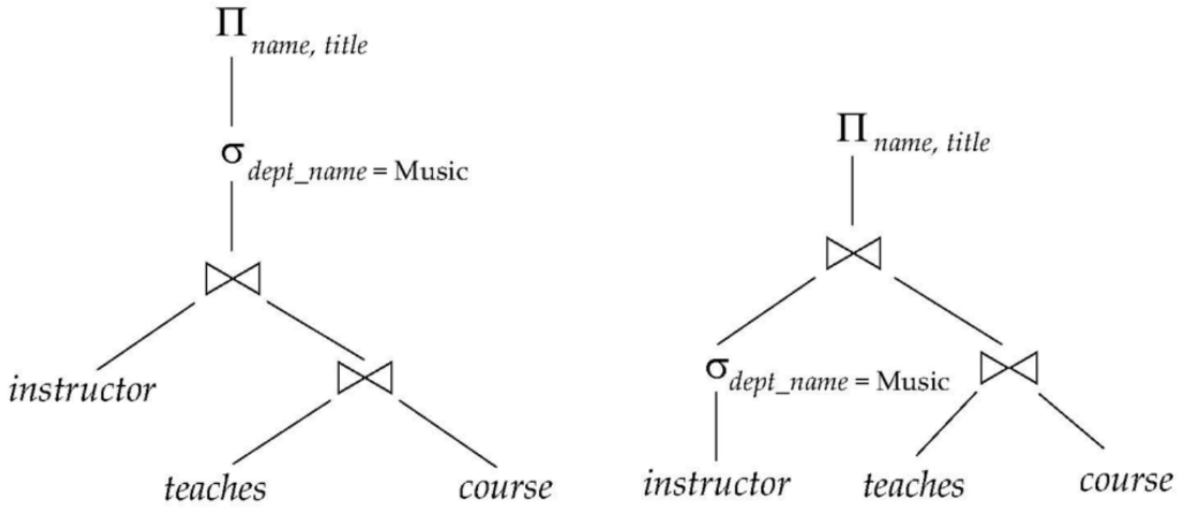


Figure 66.1: Different query evaluation plans for the same query.

The optimizer will calculate the cost of both plans and choose the one that executes faster (typically the one that reduces the dataset size earlier by performing the `Selection` operation before the `Join`).

66.2 Equivalence Rules

How does the database optimizer know that two different execution plans will return the exact same result? It relies on **Equivalence Rules**.

These mathematical rules state that certain relational algebra expressions can be transformed into other expressions without changing the final output. The optimizer applies these rules to rewrite the query into a more efficient form.

66.2.1 1. Selection Rules

- **Conjunctive selections can be split:** A selection with multiple conditions connected by AND ($\wedge$) can be broken down into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

- **Selections are commutative:** The order in which you apply filters doesn't matter.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

66.2.2 2. Projection Rules

- **Only the final projection matters:** In a sequence of projection operations, only the last one is necessary; the others can be safely omitted.

$$\Pi_{L_1}(\Pi_{L_2}(...\Pi_{L_n}(E))) = \Pi_{L_1}(E)$$

66.2.3 3. Join Rules

- **Theta-joins are commutative:** Joining Table A to Table B gives the same result as joining Table B to Table A.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

- **Natural joins are associative:** When joining three tables, it doesn't matter which two you join first.

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

- **Selections can be combined with Cartesian products to form a join:**

$$\sigma_\theta(E1 \times E2) = E_1 \bowtie_\theta E_2$$

- **Selections can be absorbed into a join condition:**

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

66.2.4 4. Set Operation Rules

- **Associative:**
  - Union: $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
  - Intersection: $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
- **Commutative:**
  - Union: $E_1 \cup E_2 = E_2 \cup E_1$
  - Intersection: $E_1 \cap E_2 = E_2 \cap E_1$
  - *(Note: Set difference is NOT commutative)*
- **Projection distributes over union:**

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

67 Architecture, Scaling, and Big Data

67.1 What do Database Applications Need?

When building data-intensive applications, you need a database that performs well under pressure. Performance is judged by three main metrics:

1. **Throughput:** Transactions per second (tps). Think of this as the number of cars passing through a toll booth every minute.
2. **Response Time:** The delay from submitting a query to getting the result. Think of this as how long one specific car waits in line.
3. **Availability:** Mean time to failure. How often is the toll booth open and working without breaking down?

67.2 RDBMS Architecture

Historically, relational databases (RDBMS) have been built using different architectures depending on the scale of the organization.

67.2.1 1. Centralized Systems

Run on a single machine (like a desktop or a single server) and do not interact with other computers.

- **Analogy:** Keeping all your company's files in a single filing cabinet in one office.

67.2.2 2. Client-Server Systems

The work is divided. Client machines handle the front-end (user interface, forms), while a powerful Server machine handles the back-end (query evaluation, transaction management, data storage).

- **Analogy:** A restaurant. The waiters (Clients) take the orders from the customers and pass them to the kitchen (Server), which does the heavy lifting of cooking the food.

67.2.3 3. Parallel Systems

Uses multiple processors and disks connected by a fast internal network. It aims for two things:

- **Speedup:** Making a single big task run faster.
- **Scaleup:** Handling a larger workload without slowing down.

67.2.4 4. Distributed Systems

Data is spread across multiple separate machines (nodes) connected via a network (like the internet).

- **Advantages:** Data can be shared globally, and if one node fails, others can keep the system running (high availability).

67.3 Scaling Databases: Horizontal vs. Vertical

When your application gets popular (like a viral social media app), your single database server will eventually hit its limit. You have two choices for scaling:

67.3.1 Vertical Scaling (Scaling Up)

This means adding more resources (CPU, RAM, faster disks) to your *existing* single machine.

- **Analogy:** You need to transport more boxes. Vertical scaling is taking your pickup truck to the mechanic and upgrading it into a massive 18-wheeler semi-truck.
- **Pros:** Cost-effective at first, less complex, no need to change your software.
- **Cons:** There is a physical limit to how big one machine can get. Eventually, you can't buy a bigger truck.

67.3.2 Horizontal Scaling (Scaling Out)

This means adding *additional* machines (nodes) to your infrastructure to share the load.

- **Analogy:** You need to transport more boxes. Horizontal scaling is buying a fleet of 10 normal pickup trucks instead of one massive semi-truck.
- **Pros:** Virtually infinite scaling, increased resilience (if one truck breaks down, you have 9 others).
- **Cons:** Much more complex. You now have to coordinate a whole fleet instead of driving just one vehicle.

67.4 The Rise of Big Data

Relational databases are fantastic, but the explosion of web applications, social media, and IoT devices generated a "perfect storm" of data that traditional RDBMS systems struggled to handle.

This is the era of **Big Data**, characterized by the **5 V's**:

1. **Volume:** Massive quantities of data (Terabytes to Exabytes).
2. **Velocity:** Data generated at lightning speed, often requiring real-time processing.
3. **Variety:** Data isn't just neat tables anymore. It's unstructured text, images, videos, and sensor logs.
4. **Variability:** Inconsistent data flows.
5. **Veracity:** Varying data quality and accuracy.

67.5 The CAP Theorem

When you use Horizontal Scaling to build a distributed database system across multiple nodes, you run into the **CAP Theorem**.

The theorem states that a distributed system can only guarantee **two out of the following three** properties at any given time:

1. **Consistency (C):** All clients see the exact same data at the same time, no matter which node they connect to.
2. **Availability (A):** The system always accepts reads and writes, even if some nodes fail.

3. **Partition Tolerance (P):** The system continues to operate even if a network failure cuts off communication between some nodes.

The ATM Analogy: Imagine two ATMs (nodes) connected to a central bank. Suddenly, the network cable is cut (a **Partition**).

- Do you let a user withdraw money anyway, risking that they might withdraw the same funds at the other ATM? (Choosing **Availability** over Consistency).
- Or do you freeze the ATMs and show an "Out of Service" error until the network is fixed, ensuring no accounts are overdrawn? (Choosing **Consistency** over Availability).

In the world of Big Data, network partitions are inevitable. Therefore, modern systems usually have to choose between **Consistency** and **Availability**.

67.6 Enter NoSQL

Because RDBMS systems rigidly enforce Consistency (ACID properties), they struggle to scale horizontally for massive, high-velocity web apps.

This led to **NoSQL (Not Only SQL)** databases. They drop strict relational rules in favor of massive scalability, often choosing Availability over strong Consistency (relying instead on *Eventual Consistency*).

67.6.1 Types of NoSQL Databases

1. **Key-Value Stores (e.g., DynamoDB, Redis)**
    - Acts like a massive dictionary. You look up a "Key" and get a "Value". Extremely fast for simple data.
    - *Example:* Storing user session data or shopping cart contents.
2. **Document Stores (e.g., MongoDB, CouchDB)**
    - Stores data in flexible, JSON-like documents. Great for hierarchical or deeply nested data.
    - *Example:* Content management systems, personalization engines.
3. **Column-Oriented Stores (e.g., Cassandra, HBase)**
    - Stores data in columns rather than rows. Excellent for searching and aggregating massive datasets.
    - *Example:* Logging user clicks on a massive e-commerce website.

4. **Graph Databases (e.g., Neo4j)**

    - Designed to store and navigate relationships (nodes and edges) rather than tables.
    - *Example:* Social networks. Finding "Friends of friends who like Pizza" is instantly resolved by walking the graph, whereas an RDBMS would require painfully slow, complex table joins.

Part XIII

Playground

68 SQL Playground

69 Python-DB Playground