Table of Contents

1 ■ 1.1 What is Data? 1

1.1 Overview 1
1.2 Real-World Example: Student Mark Sheet 2
1.3 What Can Be Data? 3
1.4 Data in Everyday Life 4
1.5 Data vs Information 4
1.6 Where Is Data Stored? 5
1.7 Data in a Database 6
1.8 Data vs Database 7
1.9 Common Beginner Mistakes 7
1.10 Placement Quick Points 8
1.11 Interview Questions 8
1.12 Practice & Hands-On Exercises 9
1.13 Key Takeaway 10

2 ■ 1.2 Data vs Information 11

2.1 Overview 11
2.2 Real-World Analogy: Exam Marks 12
2.3 Data vs Information 13
2.4 How Data Becomes Information 15
2.5 Where SQL Fits 16
2.6 Important Difference 17
2.7 Data Processing Lifecycle 18
2.8 Common Beginner Mistakes 20
2.9 Placement Quick Points 20
2.10 Interview Questions 21

2.11 Practice & Hands-On Exercises 22
2.12 Key Takeaway 24

3 ■ 1.3 Formats of Data 25
3.1 Overview 25
3.2 1. Structured Data 26
3.3 2. Semi-Structured Data 28
3.4 3. Unstructured Data 30
3.5 Simple Comparison 31
3.6 Why Does SQL Mostly Work With Structured Data? 32
3.7 Where Are These Formats Used? 32
3.8 Common Beginner Mistakes 34
3.9 Placement Quick Points 34
3.10 Interview Questions 35
3.11 Practice & Hands-On Exercises 36
3.12 Key Takeaway 38

4 ■ 1.4 Classification of Data (Qualitative & Quantitative) 39
4.1 Overview 39
4.2 Quantitative Data 40
4.3 Qualitative Data 43
4.4 Simple Comparison 45
4.5 How This Relates to SQL 46
4.6 Common Beginner Mistakes 47
4.7 Placement Quick Points 48
4.8 Interview Questions 48
4.9 Practice & Hands-On Exercises 50
4.10 Key Takeaway 52

5 ■ 2.1 Multi-Tier Architecture 53

5.1 Overview 53
5.2 Client Tier 54
5.3 Application Tier 55
5.4 Database Tier 56
5.5 How the Tiers Communicate 57
5.6 Why Use Three Tiers? 58
5.7 One-Tier, Two-Tier, and Three-Tier Architecture 59
5.8 Real-World Example: Online Banking 60
5.9 Common Beginner Mistakes 63
5.10 Placement Quick Points 64
5.11 Interview Questions 64
5.12 Practice & Hands-On Exercises 66
5.13 Key Takeaway 67

6 ■ 2.2 What is a Database? 68

6.1 Overview 68
6.2 Why Do We Need a Database? 69
6.3 Database vs Ordinary Files 70
6.4 What Is Stored Inside a Database? 71
6.5 What Can We Do With a Database? 73
6.6 How SQL Works With a Database 74
6.7 Data Persistence 75
6.8 Multi-User Access 76
6.9 Security and Access Control 77
6.10 Why Databases Are Important in Real Applications 77
6.11 Common Beginner Mistakes 79
6.12 Placement Quick Points 80
6.13 Interview Questions 80
6.14 Practice & Hands-On Exercises 82

6.15 Key Takeaway 83

7 ■ 2.3 DBMS (Database Management System) 85

7.1 Overview 85
7.2 Database vs DBMS 86
7.3 How a DBMS Works 88
7.4 Main Responsibilities of a DBMS 89
7.5 Common DBMS Examples 90
7.6 DBMS and SQL 90
7.7 Why Do We Need a DBMS? 91
7.8 Common Beginner Mistakes 92
7.9 Placement Quick Points 93
7.10 Interview Questions 93
7.11 Practice & Hands-On Exercises 95
7.12 Key Takeaway 96

8 ■ 2.4 RDBMS and the Role of SQL 97

8.1 Overview 97
8.2 What Makes an RDBMS Relational? 99
8.3 What Is a Table in an RDBMS? 100
8.4 Role of SQL 101
8.5 How SQL and an RDBMS Work Together 102
8.6 Declarative Nature of SQL 103
8.7 Real-World Example: Online Shopping 104
8.8 RDBMS vs DBMS 105
8.9 Examples of RDBMS 106
8.10 Brief History of the Relational Model 106
8.11 Common Beginner Mistakes 107
8.12 Placement Quick Points 108
8.13 Interview Questions 108

8.14 Practice & Hands-On Exercises 110
8.15 Key Takeaway 111

9 2.5 Core Database Characteristics 112
9.1 Overview 112
9.2 1. Reduced Data Redundancy 114
9.3 2. Managing Large Amounts of Data 114
9.4 3. Data Integrity 115
9.5 4. Data Security 115
9.6 5. Data Consistency 116
9.7 6. Backup and Recovery 117
9.8 7. Privacy 118
9.9 8. Data Sharing 119
9.10 Additional Database Characteristics 119
9.11 Self-Describing Nature 119
9.12 Program-Data Independence 120
9.13 Multiple Views of Data 121
9.14 Multi-User Transaction Processing 122
9.15 Common Beginner Mistakes 123
9.16 Placement Quick Points 124
9.17 Interview Questions 125
9.18 Practice & Hands-On Exercises 126
9.19 Key Takeaway 127

10 3.1 Database Transactions 128
10.1 Overview 128
10.2 Why Do We Need Transactions? 130
10.3 Transaction Commands 131
10.4 Practice Table Setup 132
10.5 Example: Successful Transaction 133

10.6 Example: Rolling Back a Transaction 133
10.7 Transaction States 134
10.8 What Happens When Something Fails? 135
10.9 Transactions and ACID 135
10.10 Real-World Example: Online Ticket Booking 136
10.11 Common Beginner Mistakes 138
10.12 Placement Quick Points 139
10.13 Interview Questions 139
10.14 Practice & Hands-On Exercises 141
10.15 Key Takeaway 142

11 • 3.2 ACID Properties 144

11.1 Overview 144
11.2 Atomicity 146
11.3 Consistency 147
11.4 Isolation 147
11.5 Durability 148
11.6 ACID in One Example 149
11.7 Practice Table 150
11.8 Example: Atomicity in Action 150
11.9 COMMIT vs ROLLBACK 151
11.10 Why ACID Matters 152
11.11 Common Beginner Mistakes 153
11.12 Placement Quick Points 154
11.13 Interview Questions 154
11.14 Practice & Hands-On Exercises 156
11.15 Key Takeaway 157

12 • 3.3 Normalization 158

12.1 Overview 158

12.2 Why Is Normalization Needed? 159
12.3 Problems in Unnormalized Data 160
12.4 Practice Table Setup 162
12.5 First Normal Form (1NF) 163
12.6 Normalized Design 164
12.7 Normalization Transformation 166
12.8 2NF and 3NF — Basic Idea 167
12.9 SQL Example 168
12.10 Benefits of Normalization 169
12.11 Common Beginner Mistakes 170
12.12 Placement Quick Points 171
12.13 Interview Questions 171
12.14 Practice & Hands-On Exercises 173
12.15 Key Takeaway 174

13 ■ 3.4 Normal Forms 176

13.1 Overview 176
13.2 First Normal Form (1NF) 177
13.3 What Does Atomic Mean? 178
13.4 Violating 1NF — Practice Table Setup 179
13.5 Satisfying 1NF 180
13.6 Second Normal Form (2NF) 181
13.7 Third Normal Form (3NF) 183
13.8 BCNF 184
13.9 Fourth Normal Form (4NF) 184
13.10 Fifth Normal Form (5NF) 185
13.11 Normal Forms at a Glance 186
13.12 Common Beginner Mistakes 187
13.13 Placement Quick Points 188

13.14 Interview Questions 188
13.15 Practice & Hands-On Exercises 190
13.16 Key Takeaway 191

14 4.1 Architecture Overview (Query Flow) 193

14.1 Overview 193
14.2 Step 1: Client Sends the Query 194
14.3 Step 2: Query Parser 195
14.4 Step 3: Query Optimizer 196
14.5 Step 4: Query Executor 197
14.6 Step 5: Storage Engine and Data Access 197
14.7 Step 6: Caching 198
14.8 Step 7: Transaction and Write Management 199
14.9 Step 8: Result Returns to the Client 200
14.10 Query Flow: Read vs Write 201
14.11 A Simple Real-World Example 202
14.12 Why Query Flow Matters for Performance 203
14.13 Common Beginner Mistakes 204
14.14 Placement Quick Points 205
14.15 Interview Questions 205
14.16 Practice & Hands-On Exercises 207
14.17 Key Takeaway 208

15 4.2 Compute Engine Layer 209

15.1 Overview 209
15.2 Network Interface — Communication Gateway 210
15.3 Query Parser 211
15.4 Optimizer 212
15.5 Query Executor 213
15.6 Execution Plan Cache 214

15.7 Complete Compute Engine Flow 215
15.8 A Simple Real-World Example 216
15.9 How the Components Work Together 217
15.10 Why the Compute Engine Matters 217
15.11 Common Beginner Mistakes 218
15.12 Placement Quick Points 219
15.13 Interview Questions 219
15.14 Practice & Hands-On Exercises 221
15.15 Key Takeaway 222

16 • 4.3 Storage Engine Layer 223

16.1 Overview 223
16.2 Access Method 224
16.3 Buffer Manager 226
16.4 Transaction Manager 227
16.5 Read and Write Paths 228
16.6 Storage Engine and Persistent Storage 229
16.7 Example: Finding an Employee 229
16.8 Example: Updating an Account 230
16.9 Storage Engine vs Compute Engine 231
16.10 Why the Storage Engine Matters for Performance 232
16.11 Common Beginner Mistakes 233
16.12 Placement Quick Points 234
16.13 Interview Questions 234
16.14 Practice & Hands-On Exercises 236
16.15 Key Takeaway 237

17 • 4.4 Physical Storage (Pages, Extents, Data Files, Database Files) 238

17.1 Overview 238
17.2 Pages 239

17.3 Extents 240
17.4 Data Files 241
17.5 Database Files 242
17.6 Transaction Log 243
17.7 How Data Moves Through Physical Storage 244
17.8 Example: Finding a Row 245
17.9 Physical Storage Hierarchy 245
17.10 Physical Storage and Query Performance 246
17.11 Common Beginner Mistakes 248
17.12 Placement Quick Points 249
17.13 Interview Questions 249
17.14 Practice & Hands-On Exercises 251
17.15 Key Takeaway 252

18 • 5.1 Overview of SQL Command Types 253

18.1 Overview 253
18.2 Quick Comparison 254
18.3 Placement Quick Points 254
18.4 Interview Questions 255
18.5 Practice & Hands-On Exercises 255
18.6 Key Takeaway 256

19 • 5.2 DDL — CREATE, DROP, TRUNCATE, ALTER 257

19.1 Overview 257
19.2 CREATE 258
19.3 ALTER 259
19.4 TRUNCATE 260
19.5 DROP 260
19.6 CREATE vs ALTER vs TRUNCATE vs DROP 261
19.7 Practice Table Setup 262

19.8 Common Beginner Mistakes 263
19.9 Placement Quick Points 264
19.10 Interview Questions 264
19.11 Practice & Hands-On Exercises 265
19.12 Key Takeaway 266

20 • 5.3 DQL — SELECT 267

20.1 Overview 267
20.2 Practice Table 268
20.3 Basic SELECT 269
20.4 SELECT Specific Columns 269
20.5 SELECT with WHERE 270
20.6 Logical Query Processing Order 270
20.7 Common Beginner Mistakes 272
20.8 Placement Quick Points 272
20.9 Interview Questions 273
20.10 Practice & Hands-On Exercises 274
20.11 Key Takeaway 275

21 • 5.4 DCL — GRANT, REVOKE 276

21.1 Overview 276
21.2 GRANT 277
21.3 REVOKE 278
21.4 Common Permissions 279
21.5 Users and Roles 279
21.6 GRANT vs REVOKE 280
21.7 Why DCL Matters 280
21.8 Common Beginner Mistakes 281
21.9 Placement Quick Points 282
21.10 Interview Questions 282

21.11 Practice & Hands-On Exercises 283
21.12 Key Takeaway 284

22 5.5 DML — INSERT, UPDATE, DELETE 286
22.1 Overview 286
22.2 Practice Table 287
22.3 INSERT 288
22.4 UPDATE 288
22.5 DELETE 289
22.6 INSERT vs UPDATE vs DELETE 290
22.7 DML and Transactions 290
22.8 Common Beginner Mistakes 292
22.9 Placement Quick Points 292
22.10 Interview Questions 293
22.11 Practice & Hands-On Exercises 294
22.12 Key Takeaway 295

23 5.6 TCL — COMMIT, ROLLBACK, SAVEPOINT 296
23.1 Overview 296
23.2 COMMIT 297
23.3 ROLLBACK 298
23.4 SAVEPOINT 298
23.5 COMMIT vs ROLLBACK vs SAVEPOINT 300
23.6 Important Difference 300
23.7 Practice Table 301
23.8 Common Beginner Mistakes 302
23.9 Placement Quick Points 302
23.10 Interview Questions 303
23.11 Practice & Hands-On Exercises 304
23.12 Key Takeaway 305

24 • 6.1 Constraints (NOT NULL, UNIQUE, PK, FK, CHECK, DEFAULT) 306

24.1 Overview 306
24.2 NOT NULL 307
24.3 UNIQUE 308
24.4 PRIMARY KEY 308
24.5 FOREIGN KEY 309
24.6 CHECK 310
24.7 DEFAULT 310
24.8 Practice Table Setup 311
24.9 FOREIGN KEY Example 312
24.10 Constraint Comparison 313
24.11 Constraints with ALTER TABLE 313
24.12 Common Beginner Mistakes 314
24.13 Placement Quick Points 315
24.14 Interview Questions 315
24.15 Practice & Hands-On Exercises 316
24.16 Key Takeaway 318

25 • 6.2 Database Objects (Table, View, Materialized View, Sequence, Function) 319

25.1 Overview 319
25.2 1. Table 320
25.3 2. View 321
25.4 3. Materialized View 322
25.5 View vs Materialized View 323
25.6 4. Sequence 323
25.7 5. Function 324
25.8 Practice Tables 325
25.9 Table vs View vs Materialized View 326

25.10 Common Beginner Mistakes 327
25.11 Placement Quick Points 328
25.12 Interview Questions 328
25.13 Practice & Hands-On Exercises 329
25.14 Key Takeaway 330

26 • 6.3 Aggregate Functions (MIN, MAX, SUM, AVG, COUNT) 331

26.1 Overview 331
26.2 Practice Tables 332
26.3 MIN() and MAX() 333
26.4 SUM() 334
26.5 AVG() 334
26.6 COUNT() 335
26.7 NULL and Aggregate Functions 336
26.8 Simple Comparison 337
26.9 Common Beginner Mistakes 338
26.10 Placement Quick Points 338
26.11 Interview Questions 339
26.12 Practice & Hands-On Exercises 340
26.13 Key Takeaway 341

27 • 6.4 Stored Procedures 342

27.1 Overview 342
27.2 PostgreSQL Example 343
27.3 Calling the Procedure 344
27.4 Why Use Stored Procedures? 345
27.5 Procedure vs Function 345
27.6 Common Beginner Mistakes 346
27.7 Placement Quick Points 347
27.8 Interview Questions 347

27.9 Practice & Hands-On Exercises 348
27.10 Key Takeaway 349

28 ■ 6.5 Triggers (BEFORE / AFTER) 350
28.1 Overview 350
28.2 BEFORE Trigger 351
28.3 AFTER Trigger 352
28.4 BEFORE vs AFTER 353
28.5 Important PostgreSQL Terms 354
28.6 Common Beginner Mistakes 355
28.7 Placement Quick Points 355
28.8 Interview Questions 356
28.9 Practice & Hands-On Exercises 357
28.10 Key Takeaway 358

29 ■ 6.6 Users, Roles & Permissions 359
29.1 Overview 359
29.2 User 360
29.3 Roles 361
29.4 Permissions 361
29.5 PostgreSQL Example 362
29.6 Common Role Types 363
29.7 Principle of Least Privilege 363
29.8 Common Beginner Mistakes 364
29.9 Placement Quick Points 364
29.10 Interview Questions 365
29.11 Practice & Hands-On Exercises 366
29.12 Key Takeaway 366

30 ■ 7.1 DISTINCT and LIMIT 367

30.1 Overview 367
30.2 Practice Table 369
30.3 DISTINCT 369
30.4 LIMIT 370
30.5 DISTINCT vs LIMIT 371
30.6 Common Beginner Mistakes 372
30.7 Placement Quick Points 372
30.8 Interview Questions 373
30.9 Practice & Hands-On Exercises 373
30.10 Key Takeaway 374

31 • 7.2 SQL Data Types (Numeric, Character, Date/Time, Special) 376

31.1 Overview 376
31.2 1. Numeric Types 377
31.3 2. Character Types 378
31.4 3. Date and Time Types 379
31.5 4. Special Types 379
31.6 Practice Table 380
31.7 Choosing the Right Data Type 381
31.8 Common Beginner Mistakes 382
31.9 Placement Quick Points 383
31.10 Interview Questions 383
31.11 Practice & Hands-Hands-On Exercises 384
31.12 Key Takeaway 385

32 • 7.3 CTAS (CREATE TABLE AS SELECT) 386

32.1 Overview 386
32.2 Syntax 387
32.3 Practice Table 388
32.4 Example: CTAS 388

32.5 What Does CTAS Copy? 389
32.6 When Is CTAS Useful? 389
32.7 CTAS vs INSERT INTO 390
32.8 Common Beginner Mistakes 391
32.9 Placement Quick Points 392
32.10 Interview Questions 392
32.11 Practice & Hands-On Exercises 393
32.12 Key Takeaway 394

33 • 7.4 Altering Tables (ADD/DROP/ALTER COLUMN) 395

33.1 Overview & Core Concepts 395
33.2 Practice Table 395
33.3 Adding a Column 395
33.4 Dropping a Column 396
33.5 Placement Quick Points 397
33.6 Interview Questions 397
33.7 Practice & Hands-On Exercises 398

34 • 8.1 The WHERE Clause 399

34.1 Overview 399
34.2 Practice Table 400
34.3 Basic WHERE 401
34.4 WHERE with a Numeric Condition 402
34.5 WHERE with Text 402
34.6 WHERE with Multiple Conditions 403
34.7 Important: NULL Values 404
34.8 WHERE vs SELECT 405
34.9 Common Beginner Mistakes 406
34.10 Placement Quick Points 407
34.11 Interview Questions 407

34.12 Practice & Hands-On Exercises 408
34.13 Key Takeaway 410

35 ■ 8.2 Comparison Operators 411
35.1 Overview 411
35.2 Practice Table 412
35.3 Examples 413
35.4 Quick Comparison 415
35.5 Important: NULL Values 415
35.6 Common Beginner Mistakes 416
35.7 Placement Quick Points 417
35.8 Interview Questions 417
35.9 Practice & Hands-On Exercises 418
35.10 Key Takeaway 420

36 ■ 8.3 Logical Operators (AND, OR, NOT) 421
36.1 Overview 421
36.2 Practice Table 422
36.3 AND 423
36.4 OR 424
36.5 NOT 424
36.6 Combining Operators 425
36.7 Common Beginner Mistakes 426
36.8 Placement Quick Points 426
36.9 Interview Questions 427
36.10 Practice & Hands-On Exercises 427
36.11 Key Takeaway 428

37 ■ 8.4 IN / NOT IN 430
37.1 Overview 430
37.2 Practice Table 431

37.3 Using IN 432
37.4 Using NOT IN 432
37.5 Critical Caveat: NOT IN with NULL Values 433
37.6 Common Beginner Mistakes 434
37.7 Placement Quick Points 434
37.8 Interview Questions 435
37.9 Practice & Hands-On Exercises 435
37.10 Key Takeaway 436

38 • 8.5 BETWEEN 437
38.1 Overview 437
38.2 Practice Table 438
38.3 BETWEEN with Numbers 439
38.4 BETWEEN with Dates 440
38.5 BETWEEN vs Comparison Operators 440
38.6 NOT BETWEEN 441
38.7 NULL Values 441
38.8 Common Beginner Mistakes 442
38.9 Placement Quick Points 442
38.10 Interview Questions 443
38.11 Practice & Hands-On Exercises 444
38.12 Key Takeaway 445

39 • 8.6 IS NULL / IS NOT NULL 446
39.1 Overview 446
39.2 Practice Table 447
39.3 IS NULL 448
39.4 IS NOT NULL 448
39.5 Why = NULL Fails (Three-Valued Logic) 449
39.6 Common Beginner Mistakes 450

39.7 Placement Quick Points 450
39.8 Interview Questions 451
39.9 Practice & Hands-On Exercises 451
39.10 Key Takeaway 452

40 8.7 LIKE / NOT LIKE and Wildcards 453

40.1 Overview 453
40.2 Practice Table 454
40.3 LIKE 455
40.4 NOT LIKE 457
40.5 Wildcard % 457
40.6 Wildcard _ 458
40.7 Common Patterns 459
40.8 LIKE vs NOT LIKE 459
40.9 Common Beginner Mistakes 460
40.10 Placement Quick Points 460
40.11 Interview Questions 461
40.12 Practice & Hands-On Exercises 462
40.13 Key Takeaway 463

41 9.1 What is a Data Model? (Conceptual / Logical / Physical) 464

41.1 Overview 464
41.2 Conceptual Data Model 465
41.3 Logical Data Model 468
41.4 Physical Data Model 470
41.5 Simple Comparison 472
41.6 One Example: Online Shopping 472
41.7 Why Data Modeling Is Important 474
41.8 Common Beginner Mistakes 475
41.9 Placement Quick Points 475

41.10 Interview Questions 476
41.11 Practice & Hands-On Exercises 477
41.12 Key Takeaway 477

42 9.2 ER Diagrams — Entities, Attributes, Relationships 478

42.1 Overview 478
42.2 Entities 479
42.3 Attributes 480
42.4 Relationships 480
42.5 ER Diagram and SQL Tables 481
42.6 Simple ER Example 483
42.7 Common Beginner Mistakes 484
42.8 Placement Quick Points 484
42.9 Interview Questions 485
42.10 Practice & Hands-On Exercises 486
42.11 Key Takeaway 487

43 9.3 Crow's Foot Notation 488

43.1 Overview 488
43.2 Example 489
43.3 Quick Memory 490
43.4 Placement Quick Points 491
43.5 Interview Questions 491
43.6 Practice & Hands-On Exercises 492
43.7 Key Takeaway 492

44 9.4 Cardinality (1:1, 1:N, M:N) 494

44.1 Overview 494
44.2 1:1 — One-to-One 495
44.3 1:N — One-to-Many 496

44.4 M:N — Many-to-Many 496
44.5 M:N Example in SQL 497
44.6 Quick Comparison 497
44.7 Common Beginner Mistakes 498
44.8 Placement Quick Points 498
44.9 Interview Questions 499
44.10 Practice & Hands-On Exercises 499
44.11 Key Takeaway 500

45 • 9.5 Mandatory vs Optional Relationships 501

45.1 Overview 501
45.2 Quick Points 502
45.3 Interview Questions 503
45.4 Practice 503
45.5 Key Takeaway 504

46 • 10.1 What are Joins? (Overview) 505

46.1 Overview 505
46.2 Simple Example 506
46.3 Quick Understanding 508
46.4 Placement Quick Points 508
46.5 Interview Questions 508
46.6 Practice & Hands-On Exercises 509
46.7 Key Takeaway 510

47 • 10.2 INNER JOIN 511

47.1 Overview 511
47.2 Practice Tables 512
47.3 Basic INNER JOIN 514
47.4 How INNER JOIN Works 515

47.5 Using Table Aliases 516
47.6 INNER JOIN vs No Match 516
47.7 Common Beginner Mistakes 517
47.8 Placement Quick Points 518
47.9 Interview Questions 518
47.10 Practice & Hands-On Exercises 519
47.11 Key Takeaway 520

48 ■ 10.3 LEFT JOIN 521
48.1 Overview 521
48.2 Practice Tables 522
48.3 Basic LEFT JOIN 523
48.4 LEFT JOIN vs INNER JOIN 524
48.5 Using Aliases 524
48.6 Finding Customers Without Orders 525
48.7 Common Beginner Mistakes 526
48.8 Placement Quick Points 526
48.9 Interview Questions 527
48.10 Practice & Hands-On Exercises 527
48.11 Key Takeaway 528

49 ■ 10.4 RIGHT JOIN 529
49.1 Overview 529
49.2 Practice Tables 530
49.3 Basic RIGHT JOIN 531
49.4 Example with No Match 531
49.5 RIGHT JOIN vs LEFT JOIN 532
49.6 Common Beginner Mistakes 533
49.7 Placement Quick Points 534
49.8 Interview Questions 534

49.9 Practice & Hands-On Exercises 535
49.10 Key Takeaway 536

50 10.5 FULL JOIN 537
50.1 Overview 537
50.2 Practice Tables 538
50.3 Basic FULL JOIN 539
50.4 Understanding the Result 539
50.5 FULL JOIN vs Other Joins 540
50.6 Important Note About the Extra Order 541
50.7 Finding Unmatched Rows 541
50.8 Common Beginner Mistakes 542
50.9 Placement Quick Points 543
50.10 Interview Questions 543
50.11 Practice & Hands-On Exercises 544
50.12 Key Takeaway 545

51 10.6 CROSS JOIN 546
51.1 Overview 546
51.2 Practice Tables 547
51.3 Basic CROSS JOIN 548
51.4 CROSS JOIN vs INNER JOIN 549
51.5 Common Beginner Mistake 549
51.6 Placement Quick Points 550
51.7 Interview Questions 550
51.8 Practice & Hands-On Exercises 551
51.9 Key Takeaway 551

52 10.7 SELF JOIN 552
52.1 Overview 552

52.2 Practice Table 554
52.3 Example: Employee and Manager 554
52.4 How SELF JOIN Works 555
52.5 SELF JOIN vs Normal JOIN 556
52.6 Common Beginner Mistakes 557
52.7 Placement Quick Points 557
52.8 Interview Questions 558
52.9 Practice & Hands-On Exercises 558
52.10 Key Takeaway 559

53 • 11.1 GROUP BY 560

53.1 Overview 560
53.2 Practice Table 561
53.3 Basic GROUP BY 562
53.4 GROUP BY with Other Aggregates 563
53.5 How GROUP BY Works 563
53.6 GROUP BY vs SELECT 564
53.7 Common Beginner Mistakes 565
53.8 Placement Quick Points 565
53.9 Interview Questions 566
53.10 Practice & Hands-On Exercises 567
53.11 Key Takeaway 568

54 • 11.2 HAVING (vs WHERE) 569

54.1 Overview 569
54.2 Basic HAVING 570
54.3 WHERE vs HAVING 571
54.4 Important Point 572
54.5 Simple Flow 572
54.6 Common Beginner Mistakes 573

54.7 Placement Quick Points 573
54.8 Interview Questions 574
54.9 Practice & Hands-On Exercises 575
54.10 Key Takeaway 576

55 ■ 11.3 UNION / UNION ALL 577
55.1 Overview 577
55.2 Practice Tables 578
55.3 UNION 579
55.4 UNION ALL 580
55.5 Important Rule 581
55.6 UNION vs UNION ALL 581
55.7 Common Beginner Mistakes 582
55.8 Placement Quick Points 582
55.9 Interview Questions 583
55.10 Practice & Hands-On Exercises 583
55.11 Key Takeaway 584

56 ■ 11.4 INTERSECT / EXCEPT 586
56.1 Overview 586
56.2 Practice Tables 587
56.3 INTERSECT 587
56.4 EXCEPT 588
56.5 Simple Comparison 589
56.6 Important Rule 589
56.7 Common Beginner Mistakes 590
56.8 Placement Quick Points 590
56.9 Interview Questions 591
56.10 Practice & Hands-On Exercises 591
56.11 Key Takeaway 592

57 • 11.5 Subqueries (WHERE, FROM, SELECT) 593

57.1 Overview 593
57.2 Subquery in WHERE 594
57.3 Subquery with IN 595
57.4 Three Common Places 596
57.5 Subquery in FROM 596
57.6 Subquery in SELECT 597
57.7 Common Beginner Mistakes 598
57.8 Placement Quick Points 598
57.9 Interview Questions 599
57.10 Practice & Hands-On Exercises 600
57.11 Key Takeaway 601

58 • 12.1 Window Functions Overview (OVER, PARTITION BY, ORDER BY) 602

58.1 Overview 602
58.2 The OVER Clause 603
58.3 PARTITION BY 604
58.4 ORDER BY 605
58.5 Simple PostgreSQL Example 605
58.6 GROUP BY vs Window Function 606
58.7 Common Beginner Mistakes 608
58.8 Placement Quick Points 608
58.9 Interview Questions 609
58.10 Practice & Hands-On Exercises 610
58.11 Key Takeaway 611

59 • 12.2 RANK() and DENSE_RANK() 612

59.1 Overview 612
59.2 Practice Table 613

59.3 RANK() 614
59.4 DENSE_RANK() 615
59.5 RANK vs DENSE_RANK 616
59.6 Common Beginner Mistakes 617
59.7 Placement Quick Points 617
59.8 Interview Questions 618
59.9 Practice & Hands-On Exercises 618
59.10 Key Takeaway 619

60 ■ 12.3 ROW_NUMBER() 620

60.1 Overview 620
60.2 Practice Table 621
60.3 Example: ROW_NUMBER() 621
60.4 How It Works 622
60.5 RANK vs DENSE_RANK vs ROW_NUMBER 623
60.6 Common Uses 624
60.7 Common Beginner Mistakes 625
60.8 Placement Quick Points 625
60.9 Interview Questions 626
60.10 Practice & Hands-On Exercises 626
60.11 Key Takeaway 627

61 ■ 12.4 LAG() 629

61.1 Overview 629
61.2 Syntax 630
61.3 Practice Table 631
61.4 Example: LAG() 631
61.5 How LAG() Works 632
61.6 Comparing Current and Previous Values 633
61.7 Common Beginner Mistakes 634

61.8 Placement Quick Points 634
61.9 Interview Questions 635
61.10 Practice & Hands-On Exercises 635
61.11 Key Takeaway 636

62 12.5 Table Partitioning (RANGE, LIST, HASH) 637
62.1 Overview 637
62.2 RANGE Partitioning 638
62.3 LIST Partitioning 639
62.4 HASH Partitioning 639
62.5 Partition Pruning 640
62.6 RANGE vs LIST vs HASH 641
62.7 Common Beginner Mistakes 642
62.8 Placement Quick Points 642
62.9 Interview Questions 643
62.10 Practice & Hands-On Exercises 644
62.11 Key Takeaway 644

1.1 What is Data?

Level: 1 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Before learning SQL and databases, we first need to understand **data**.

Data is the basic information that computer systems **collect, store, process, and use**.

Examples from a college:

```text
Student Name
Student ID
Age
Department
Marks
Attendance
Email
```

> *Data is a collection of facts or values that can be stored, processed, and used to obtain useful information.*

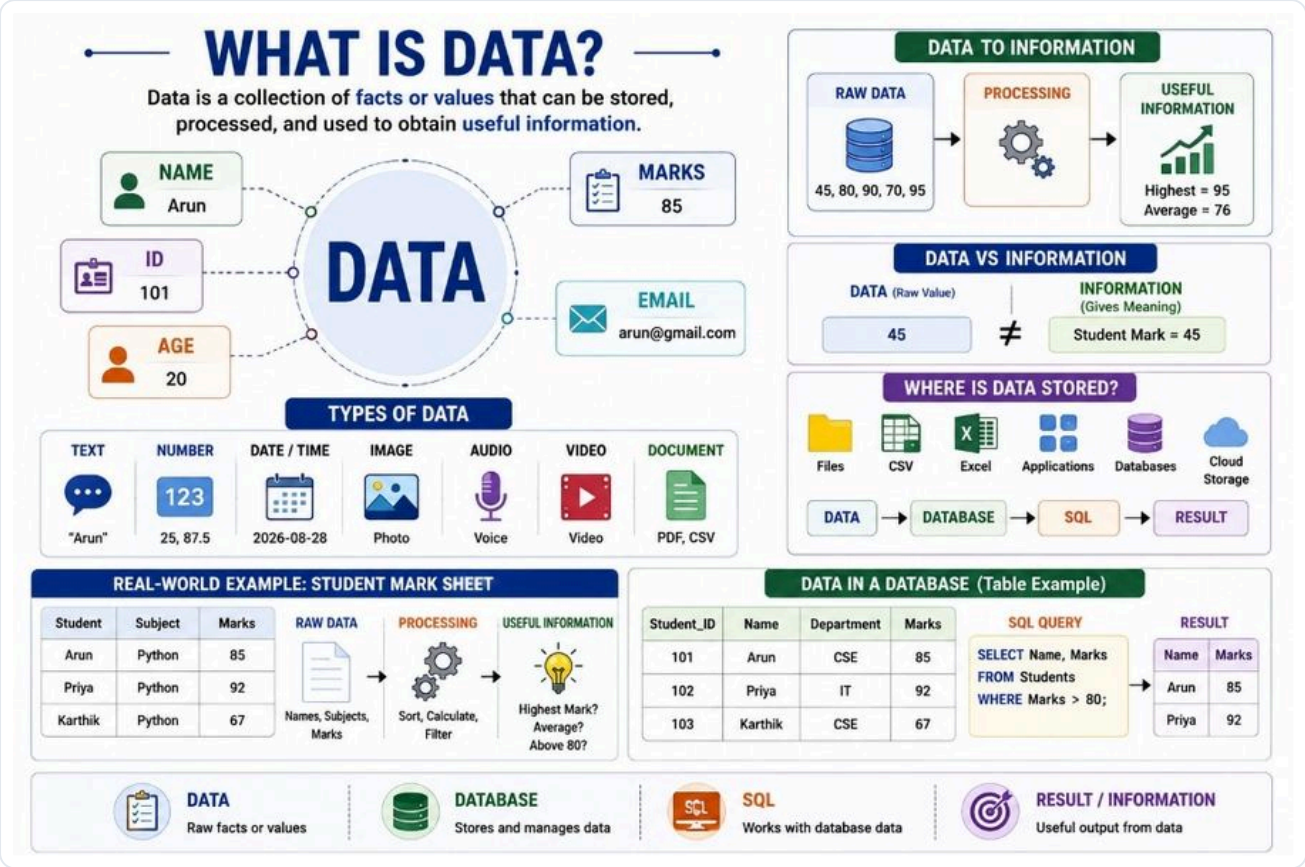


Figure 1: Data represents facts and values collected from real-world activities and stored or processed by computer systems.

Real-World Example: Student Mark Sheet

Consider a college mark sheet:

Student	Subject	Marks
Arun	Python	85
Priya	Python	92
Karthik	Python	67

Values such as `Arun`, `Python`, and `85` are **data**.

When we process the data, we can answer:

```text
Highest mark?
Class average?
Who scored above 80?
```

```text
RAW DATA
   ↓
PROCESSING
   ↓
USEFUL INFORMATION
```

What Can Be Data?

Data can have many forms:

Type	Example
Text	`"Arun"`
Numeric	`25`, `87.5`
Date/Time	`2026-08-28`
Image	Student photo
Audio	Voice recording
Video	Online class
Document/File	PDF, CSV

Data in Everyday Life

When you order food, an application may collect:

```text
Customer Name
Restaurant
Food Item
Quantity
Price
Delivery Address
Order Time
Payment Status
```

The system can process this data into:

```text
Order Total = ₹450
Payment Status = Successful
Delivery Status = Out for Delivery
```

Data vs Information

Data is a raw value.

```text
45
```

Information gives that value meaning.

Example:

```
TEXT

Raw Data:
45, 80, 90, 70, 95

After Processing:
Highest Mark = 95
Average Mark = 76
```

Data → Raw value
Information → Meaningful processed data

Where Is Data Stored?

Data can exist in:

```
TEXT

Files
CSV
Excel
Applications
Databases
Cloud Storage
```

When data becomes large and needs organized storage and querying, databases become useful.

Data in a Database

A database table organizes data into rows and columns:

Student_ID	Name	Department	Marks
101	Arun	CSE	85
102	Priya	IT	92
103	Karthik	CSE	67

SQL can then retrieve required data:

```sql
SELECT Name, Marks
FROM Students
WHERE Marks > 80;
```

Result:

Name	Marks
Arun	85
Priya	92

Data vs Database

```text
DATA
→ Actual values

DATABASE
→ Organized system for storing and managing those values
```

Copy

> *Data is the content; a database organizes and manages the content.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Data means only numbers	Data can be text, numbers, dates, images, audio, video, and more
Data and information are the same	Information gives data useful meaning
Data exists only inside databases	Data can exist in files, applications, documents, and memory
A database is the same as data	A database stores and manages data
A single value always has meaning	Context may be needed

Placement Quick Points

```text
DATA        → RAW FACTS / VALUES
INFORMATION → MEANINGFUL PROCESSED DATA
DATABASE    → STORES AND MANAGES DATA
SQL         → WORKS WITH DATABASE DATA
```

- Data is the basic input used by computer systems.
- Data can exist in many forms.
- Processing and context can turn data into useful information.
- A database organizes and manages data.
- SQL is used to work with data stored in relational databases.

Interview Questions

Q1. What is data?

Data is a collection of raw facts or values that can be stored and processed.

Q2. Give examples of data.

`text Name → Arun Age → 20 Marks → 85 Department → CSE`

Images, audio, video, and documents can also be data.

Q3. What is the difference between data and information?

`text Data → Raw facts Information → Meaningful processed data`

Q4. Can data exist without a database?

> Yes. It can exist in files, CSVs, Excel, applications, documents, memory, and other storage systems.

Q5. What is the relationship between data, database, and SQL?

> `text DATA ↓ DATABASE ↓ SQL ↓ RESULT`

Practice & Hands-On Exercises

1. Define data in your own words.
2. Give five examples of college data.
3. Give three examples of data collected by a food delivery app.
4. Explain data vs information with your own example.
5. Explain:

```
CODE                                                    Copy

Data → Database → SQL → Result
```

1. Identify three pieces of data from a student table.
2. Explain why `100` may need context before it becomes meaningful information.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
DATA
↓
RAW FACTS / VALUES
↓
DATABASE
↓
SQL
↓
USEFUL RESULT / INFORMATION
```

> *Data is the foundation of databases. SQL is one of the main tools used to work with that data.*

End of What is Data?

1.2 Data vs Information

Level: 1 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Data and **Information** are closely related, but they are not the same.

Data is a raw fact or value. By itself, it may not give us enough meaning.

Information is data that has been **processed, organized, or given context** so that it becomes meaningful and useful.

The basic idea is:

```
TEXT
DATA → PROCESSING → INFORMATION
```

For example:

```
TEXT
80
```

This is data. We do not know what `80` represents.

After adding context:

```
TEXT
Pugal scored 80 marks in Maths.
```

Now the value has meaning. It has become information.

> *Data is the raw input. Information is meaningful output obtained from data.*

Real-World Analogy: Exam Marks

Imagine a teacher has these values:

```
TEXT
Copy

80, 72, 91, 65, 88
```

These are **data**.

The teacher processes them and finds:

```
TEXT
Copy

Highest Mark = 91
Average Mark = 79.2
```

These results provide useful information.

The same data can answer different questions depending on how it is processed.

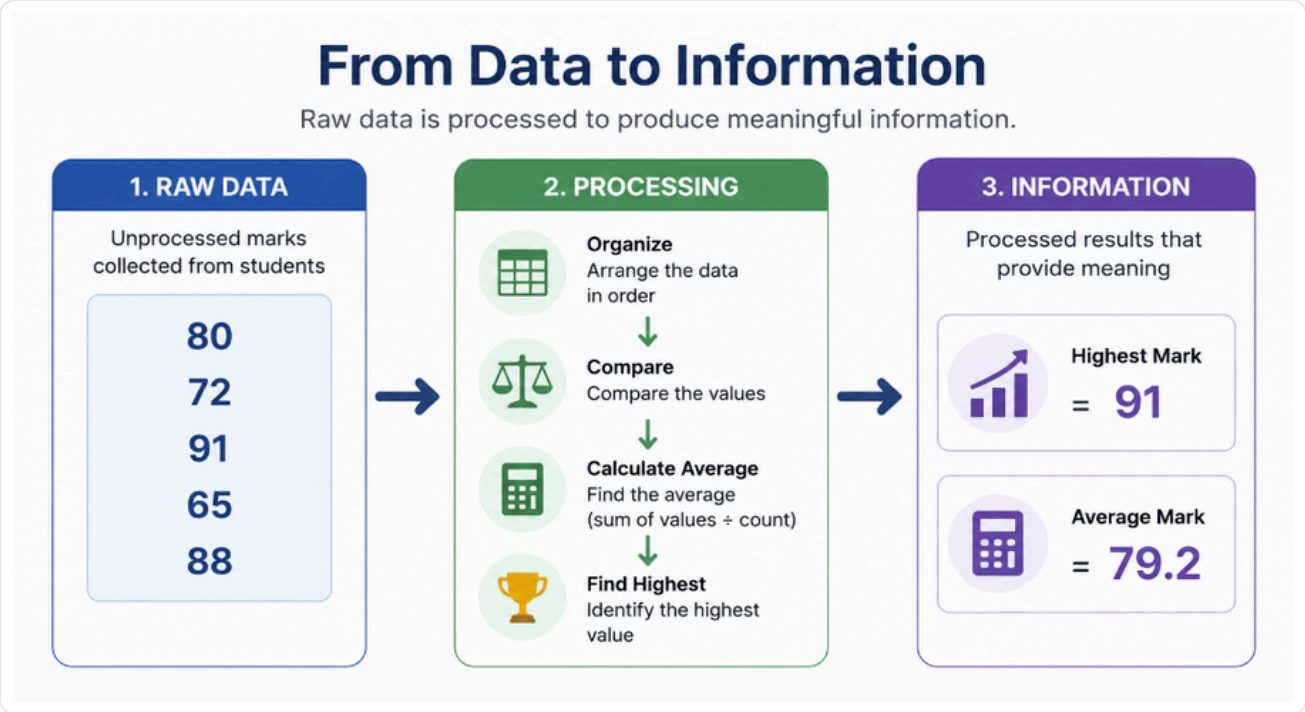


Figure 1: Raw data becomes meaningful information when it is processed and given context.

Data vs Information

Data

Data is a raw fact, value, or observation.

Examples:

```text
80
Pugal
Maths
2026-03-01
```

These values may not provide enough meaning by themselves.

Information

Information is data that has been processed or given context so that it becomes meaningful.

Example:

```text
Pugal scored 80 marks in Maths.
```

Now we know:

- **Who** → Pugal
- **What** → scored
- **Value** → 80
- **Subject** → Maths

That additional context makes the data useful.

Simple Comparison

Data	Information
Raw fact or value	Meaningful result
May lack context	Has useful context
Used as input	Used for understanding and decisions
Example: `80`	Example: `Pugal scored 80 in Maths`

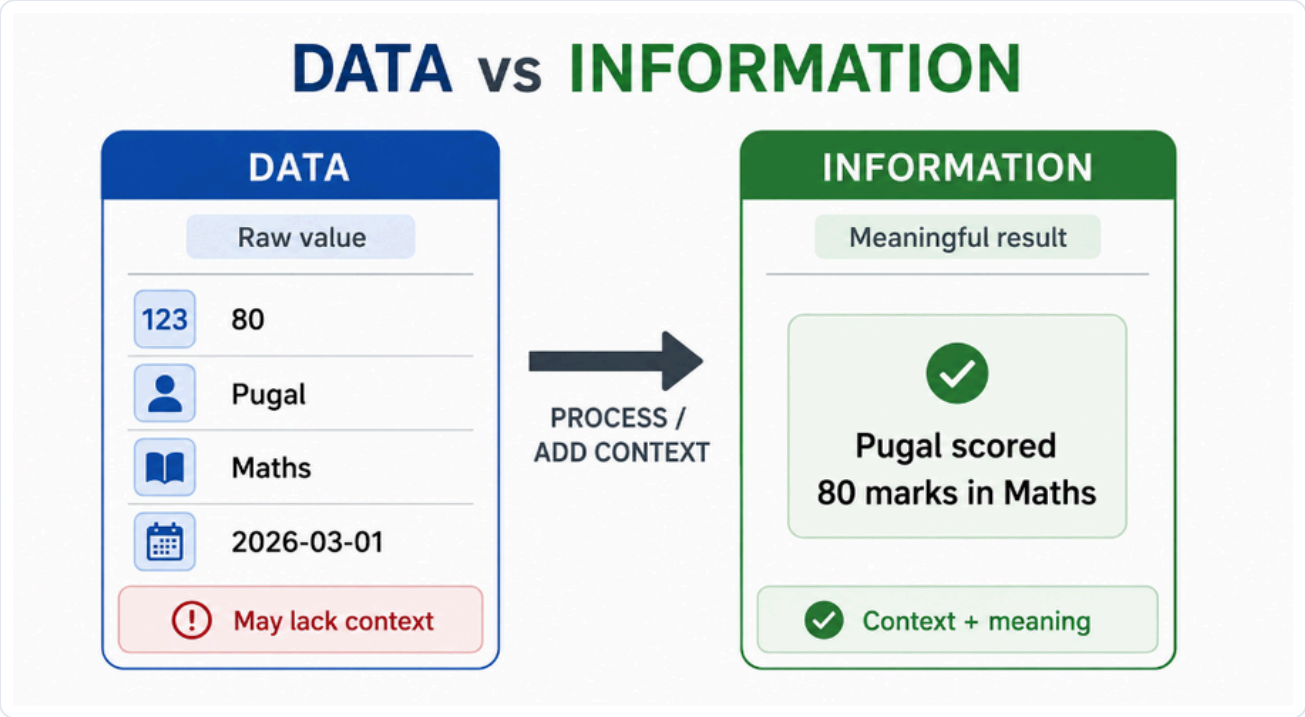


Figure 2: Data becomes information when it is processed or given meaningful context.

How Data Becomes Information

Data does not automatically become useful information just because it is stored.

It usually needs to be **processed**.

For example, a store may collect:

```text
Product     Quantity     Date
Laptop      5            2026-03-01
Phone       8            2026-03-01
Laptop      3            2026-03-02
```

This is data.

After processing, the business can determine:

```text
Total Laptops Sold = 8
Total Phones Sold  = 8
```

These results are information because they help the business understand its sales.

A simple data-processing flow is:

```text
DATA COLLECTION
       ↓
PROCESSING
       ↓
STORAGE
       ↓
RESULT / INFORMATION
```

In SQL-based systems, SQL is mainly used to **retrieve, filter, join, sort, group, and aggregate stored data** so that useful results can be produced.

Where SQL Fits

Consider a database containing sales data.

SQL can be used to ask questions such as:

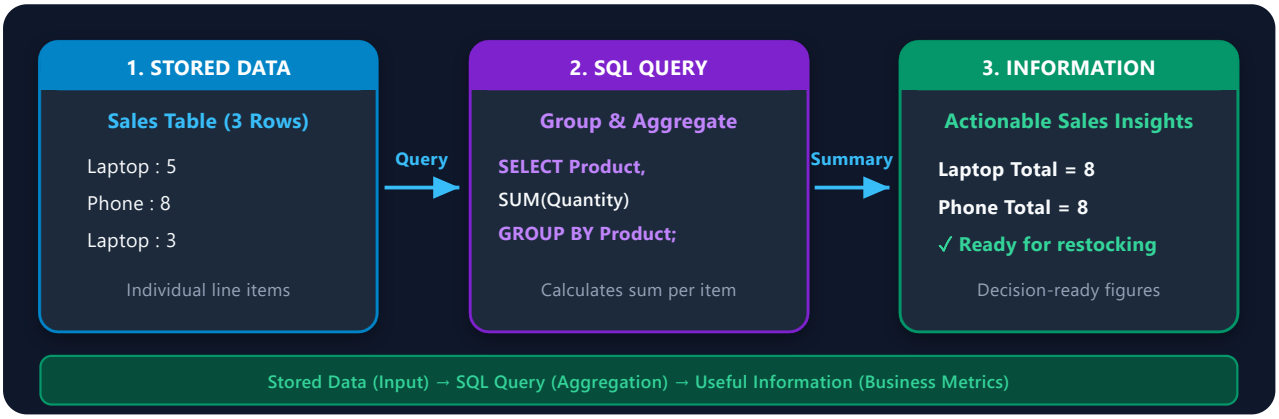
```sql
SELECT Product, SUM(Quantity) AS Total_Sold
FROM Sales
GROUP BY Product;
```

The database may produce:

Product	Total_Sold
Laptop	8
Phone	8

The original sales records are **data**.

The summarized result gives us useful **information**.



SQL queries transform raw database table rows into concise, summarized business information.

Important Difference

A common misunderstanding is:

> *"Data becomes information simply by being stored in a database."*

Not necessarily.

A database mainly helps us **store and manage data**.

Data becomes useful information when we **interpret it, process it, summarize it, or give it context**.

For example:

```TEXT
Database stores:
85
92
78
90
```

A query can calculate:

```TEXT
Average Mark = 86.25
```

The average is more meaningful for analysis.

Data Processing Lifecycle

A simplified data-processing flow can be understood as:

1. Data Collection

Data is collected from users, applications, sensors, transactions, and other sources.

2. Processing

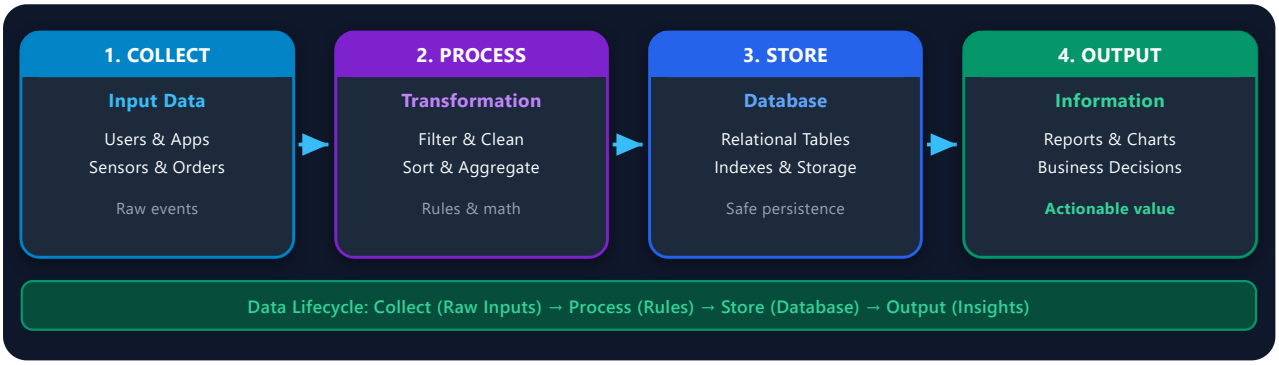
Data may be filtered, sorted, grouped, calculated, or combined.

3. Storage

The data is stored in a suitable system such as a database.

4. Output

Processed results can be shown through reports, dashboards, applications, or other outputs.



The 4-stage data processing lifecycle: from raw collection to business decision outputs.

SQL is especially useful when working with **structured data in relational databases**, particularly for querying and transforming that data into useful results.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Data and information are the same	Data is raw; information provides meaningful context or results
Every number is information	A number may need context to become meaningful
Storing data automatically makes it information	Storage and interpretation/processing are different
More data always means more useful information	Useful information depends on context and processing
SQL is only used to store data	SQL is also used to retrieve, filter, join, group, aggregate, insert, update, and delete data
Information must always be a sentence	Information can also be a number, table, report, chart, or other meaningful result

Placement Quick Points

```text
DATA        → RAW FACTS OR VALUES
PROCESSING  → ORGANIZING / FILTERING / CALCULATING / ANALYZING
INFORMATION → MEANINGFUL RESULT WITH CONTEXT
SQL         → USED TO QUERY AND PROCESS DATA IN RELATIONAL DATABASES
```

- Data is the raw input.
- Information is meaningful data with context or processing.

- `80` is data; `Pugal scored 80 marks in Maths` is information.
- Data can be processed using filtering, sorting, grouping, aggregation, and other operations.
- A database stores and manages data; processing and interpretation help turn it into useful information.
- SQL is commonly used to retrieve and transform relational database data into useful results.

Interview Questions

Q1. What is the difference between data and information?

Data is a raw fact or value, while **information** is data that has been processed or given context to make it meaningful.

Example:

```text Data: 80

Information: Pugal scored 80 marks in Maths. ```

**Q2. Give a real-world example of data becoming information.**

**R**aw sales data:

`text Laptop = 5 Laptop = 3`

**A**fter processing:

`text Total Laptops Sold = 8`

**T**he total provides useful information.
```

Q3. What are the basic stages of the data-processing lifecycle?

A simplified lifecycle is:

```
text Data Collection ↓ Processing ↓ Storage ↓ Output
```

Q4. Where does SQL fit into the data-processing process?

SQL is commonly used to query and process data stored in relational databases. It can filter, sort, join, group, and aggregate data to produce useful results.

Q5. Does storing data in a database automatically make it information?

No. A database stores and manages data. Data becomes useful information when it is processed, interpreted, summarized, or given meaningful context.

Q6. Why is context important?

Context tells us what a value represents.

For example:

```
text 45
```

does not tell us whether it means marks, age, price, or quantity.

But:

```
text Student scored 45 marks.
```

has clear meaning.

Practice & Hands-On Exercises

Beginner Practice

1. Define data in your own words.
2. Define information in your own words.
3. Explain the difference between data and information using a college example.
4. Give three examples where raw data can be converted into useful information.

Think and Answer

1. Why is `100` not meaningful by itself?
2. Can the same data produce different information? Give an example.
3. Does storing data in a database automatically make it information? Explain.
4. Explain the basic data-processing lifecycle.

SQL Connection

Consider:

Product	Quantity	Date
Laptop	5	2026-03-01
Phone	8	2026-03-01
Laptop	3	2026-03-02

1. Identify the raw data in the table.
2. How many laptops were sold in total?
3. Which result would be useful information for the store owner?
4. Write a SQL query that calculates the total quantity sold for each product:

```text
SELECT Product, SUM(Quantity) AS Total_Sold
FROM Sales
GROUP BY Product;
```

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```TEXT
RAW DATA
    ↓
PROCESSING + CONTEXT
    ↓
MEANINGFUL INFORMATION
```

> ***Data is the raw material. Information is the meaningful result we obtain by processing and understanding that data.***

End of Data vs Information

1.3 Formats of Data

Level: 1 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Data can be organized in different ways depending on how clearly its structure is defined.

The three common formats are:

1. **Structured Data**
2. **Semi-Structured Data**
3. **Unstructured Data**

The easiest way to understand them is:

```text
Structured      → Fixed structure
Semi-Structured → Some structure
Unstructured    → No fixed structure
```

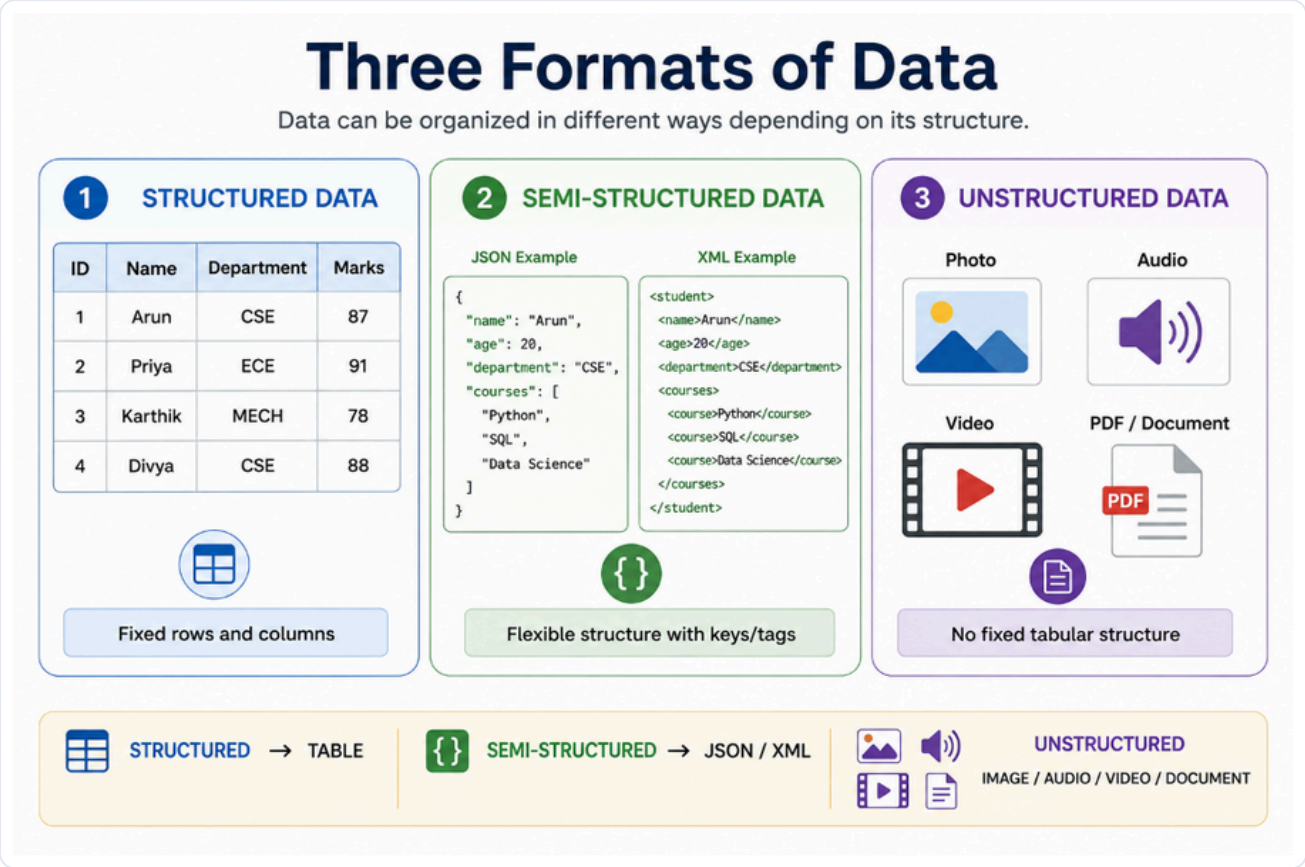


Figure 1: Data can be structured, semi-structured, or unstructured depending on how its structure is defined.

1. Structured Data

Structured data is data organized in a fixed format, usually in **rows and columns**.

It follows a defined structure, which makes it easy to store, search, sort, and analyze.

Structured data is commonly used in **relational databases**.

Example: Student Marks Table

ID	Name	English	Maths	Social Science	Total
1	Marilyn	38	24	28	90
2	John	41	24	32	97
3	Joseph	33	39	45	117
4	Daniel	35	36	25	96
5	Anthony	31	30	43	104

In this table:

- A **row** represents one complete record.
- A **column** represents one attribute.
- Every row follows the same structure.
- Each value has a defined position and meaning.

For example:

```text
Row    → One student's record
Column → One attribute such as Name or Maths
```

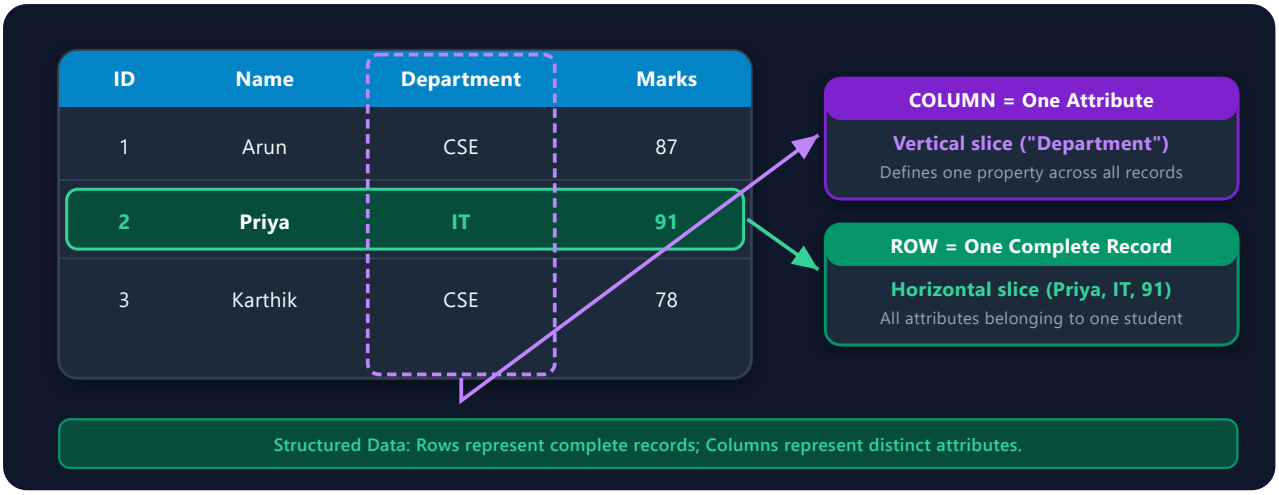


Figure 2: In structured data, rows represent records and columns represent attributes.

Real-World Examples

- Student records
- Employee records
- Product information
- Bank transactions
- Customer orders

2. Semi-Structured Data

Semi-structured data has some organization, but it does not have the rigid rows-and-columns structure of a relational table.

It often uses **keys, tags, or nested structures** to describe the data.

Common examples include:

- JSON
- XML

- YAML
- Log data

Example: JSON

```JSON
{
  "name": "John Smith",
  "age": 20,
  "department": "CSE",
  "courses": [
    "Python",
    "SQL"
  ]
}
```

Here:

- `"name"` is a key.
- `"John Smith"` is its value.
- `"courses"` contains multiple values.
- Data can be nested inside other data.

Unlike a relational table, different JSON objects can contain different fields.

For example:

```JSON
{
  "name": "John",
  "age": 20
}
```

Another object might contain:

The structure is still understandable, but it is more flexible than a fixed table.

3. Unstructured Data

Unstructured data does not follow a predefined tabular structure.

Examples include:

- Photos
- Audio recordings
- Videos
- PDF documents
- Scanned documents
- Voice recordings

For example:

```text
student_photo.jpg
lecture.mp4
voice_note.mp3
resume.pdf
```

These files contain useful information, but they do not naturally fit into fixed rows and columns like a student table.

Simple Comparison

Feature	Structured	Semi-Structured	Unstructured
Structure	Fixed	Flexible	No fixed structure
Common format	Rows and columns	Keys, tags, nested fields	Files / free-form content
Examples	SQL tables, CSV	JSON, XML, YAML	Images, audio, video, PDFs
Schema	Usually predefined	Flexible or partially defined	No fixed tabular schema
Easy to query as tables	Yes	Depends on the system	Usually requires specialized processing

Key Idea

```text
STRUCTURED
     ↓
FIXED TABLE STRUCTURE

SEMI-STRUCTURED
     ↓
FLEXIBLE STRUCTURE

UNSTRUCTURED
     ↓
NO FIXED TABULAR STRUCTURE
```

Why Does SQL Mostly Work With Structured Data?

SQL was designed primarily for working with **structured data in relational databases**.

For example, a database may contain:

```text
Students
Orders
Products
Employees
```

Each table has defined columns and rows.

SQL can then be used to retrieve and analyze that data.

For example:

```sql
SELECT Name, Marks
FROM Students
WHERE Marks > 80;
```

This works naturally because the student data has a defined table structure.

Modern relational databases can also support semi-structured values such as JSON, but the relational table remains a core SQL data model.

Where Are These Formats Used?

Structured Data

Common in:

- Banking systems
- College systems
- Payroll systems
- Order management
- Transaction processing

Semi-Structured Data

Common in:

- Web APIs
- Application configuration
- Event data
- Application logs
- Data exchange between systems

Unstructured Data

Common in:

- Social media
- Video platforms
- Digital documents
- CCTV systems
- Customer support recordings

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Structured data means only SQL data	Structured data can also exist in formats such as CSV and spreadsheets
Semi-structured data has no structure	It has some structure through keys, tags, nesting, or metadata
JSON is the same as a relational table	JSON has a flexible structure and can contain nested data
Unstructured data is useless	Unstructured data can contain valuable business and personal information
SQL can work only with structured data	SQL primarily works with relational/structured data, and modern databases can also support some semi-structured data

Placement Quick Points

```text
STRUCTURED
→ FIXED ROWS AND COLUMNS
→ EXAMPLE: RELATIONAL TABLE

SEMI-STRUCTURED
→ KEYS / TAGS / NESTED STRUCTURE
→ EXAMPLE: JSON, XML

UNSTRUCTURED
→ NO FIXED TABULAR STRUCTURE
→ EXAMPLE: IMAGE, AUDIO, VIDEO, PDF
```

- **Structured data** follows a defined schema and is commonly stored in relational tables.
- **Semi-structured data** has organization through keys, tags, or nested structures but is more flexible than a table.
- **Unstructured data** does not follow a fixed tabular structure.
- SQL is primarily used with structured relational data.
- Modern database systems can also work with some semi-structured data, especially JSON.

Interview Questions

Q1. What is structured data?

Structured data is data organized in a predefined format, usually using rows and columns in a table.

Q2. What is semi-structured data?

Semi-structured data has some organization through keys, tags, metadata, or nested fields, but it does not follow a rigid table structure.

Q3. What is unstructured data?

Unstructured data does not follow a predefined tabular structure. Examples include images, videos, audio files, and documents.

Q4. What is the main difference between structured and semi-structured data?

Structured data follows a predefined schema, usually rows and columns. Semi-structured data is more flexible and may use keys, tags, and nested structures.

Q5. Is JSON structured or semi-structured?

JSON is generally considered **semi-structured data** because it uses keys and values and can contain nested structures without requiring a fixed table schema.

Q6. Give examples of unstructured data.

Examples include:

`text Images Videos Audio recordings PDF documents Scanned documents`

Q7. Why is structured data suitable for relational databases?

Relational databases organize data into tables with predefined columns and rows, which matches the structure of structured data.

Practice & Hands-On Exercises

Beginner Practice

1. Define structured data in your own words.
2. Define semi-structured data in your own words.
3. Define unstructured data in your own words.
4. Give three examples of each format.
5. Explain the difference between a row and a column.

Classify the Data

Classify each as **Structured, Semi-Structured, or Unstructured**:

1. Student table in MySQL
2. JSON response from a web API
3. MP3 audio file
4. Employee CSV file
5. XML document
6. Video recording
7. PDF document

Think and Answer

1. Why is JSON more flexible than a relational table?
2. Why is a student database table considered structured data?
3. Why can an image not naturally be represented as rows and columns like a student table?

SQL Connection

Consider:

```text
Student Table
ID | Name | Department | Marks
```

1. Why is this structured data?
2. Which part represents the records?
3. Which part represents the attributes?
4. Write a SQL query to display students who scored more than 80.

```text
SELECT ID, Name, Department, Marks
FROM Students
WHERE Marks > 80;
```

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```SQL
STRUCTURED
→ FIXED TABLE
→ ROWS + COLUMNS

SEMI-STRUCTURED
→ FLEXIBLE STRUCTURE
→ KEYS / TAGS / NESTED DATA

UNSTRUCTURED
→ NO FIXED TABULAR STRUCTURE
→ FILES / MEDIA / DOCUMENTS
```

> *Structured data has a fixed format, semi-structured data has a flexible format, and unstructured data has no fixed tabular structure.*

End of Formats of Data

1.4 Classification of Data (Qualitative & Quantitative)

Level: 1 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Data can be classified not only by its format, but also by **what the data represents**.

There are two main categories:

```text
DATA
│
├── QUALITATIVE
│   ├── Nominal
│   └── Ordinal
│
└── QUANTITATIVE
    ├── Discrete
    └── Continuous
```

The easiest way to remember the difference:

> *Qualitative data describes a quality or category. Quantitative data represents a quantity that can be counted or measured.*

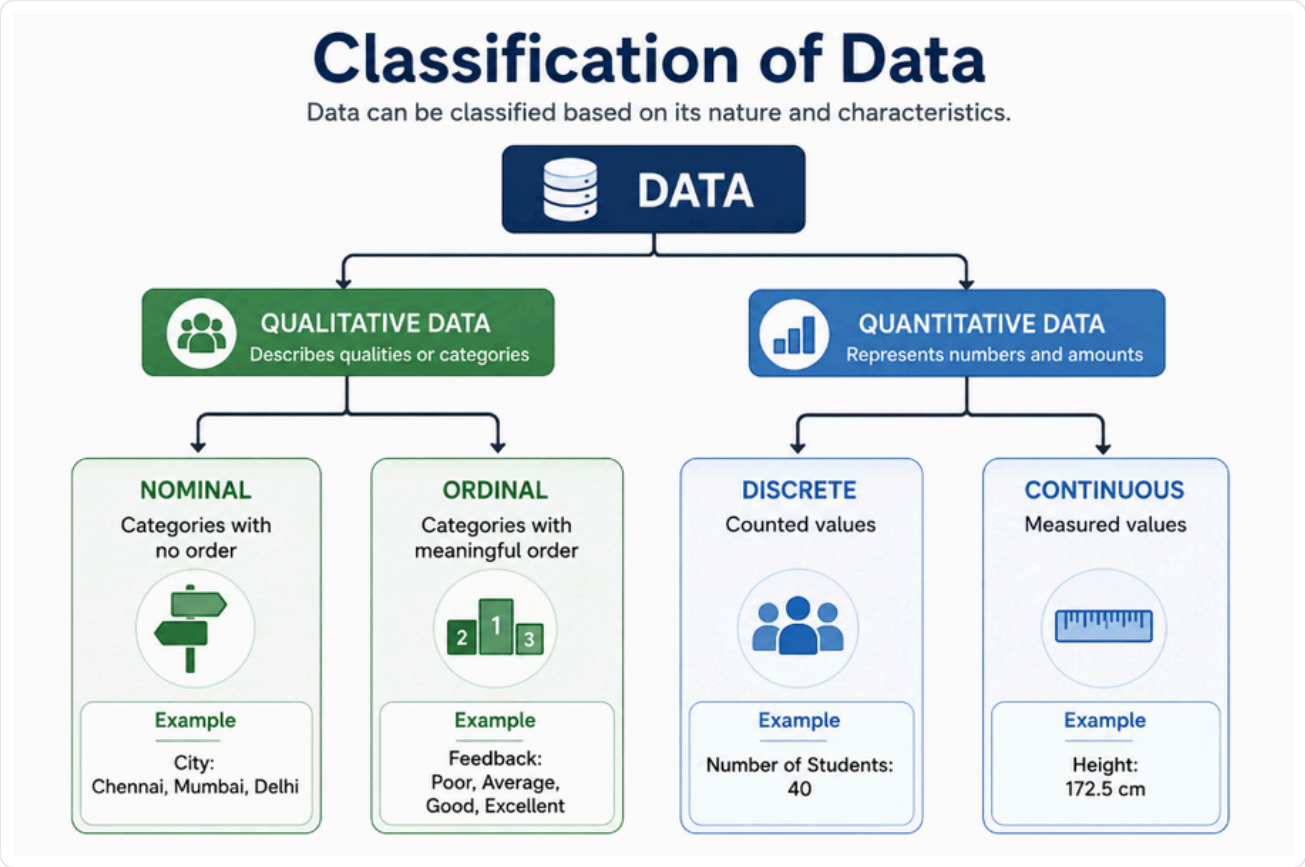


Figure 1: Data can be classified as qualitative or quantitative, with each category having two common subtypes.

Quantitative Data

Quantitative data represents values that can be **counted or measured using numbers**.

It usually answers questions such as:

- How many?
- How much?
- How often?

Examples:

```text
Number of students = 40
Orders placed = 125
Temperature = 36.7°C
Height = 172.5 cm
```

Quantitative data has two common types: **Discrete** and **Continuous**.

Discrete Data

Discrete data consists of separate, countable values.

It is usually obtained by **counting**.

Examples:

- Number of students in a class
- Number of website clicks
- Number of cars in a parking lot
- Number of orders

For example:

```text
Students = 40
```

You can have 40 students, but not 40.5 students.

> ***Discrete data is counted and normally represented using separate whole-number values.***

Continuous Data

Continuous data is data that can take different values within a range.

It is usually obtained by **measuring**.

Examples:

- Height
- Weight
- Temperature
- Time
- Distance

For example:

```text
Height = 172.5 cm
Temperature = 36.7°C
```

A measurement can often be made more precise:

```text
172 cm
172.5 cm
172.53 cm
172.531 cm
```

> ***Continuous data is measured and can take fractional or decimal values within a range.***

Qualitative Data

Qualitative data describes qualities, characteristics, or categories rather than quantities.

It answers questions such as:

> *"What kind?"*

Examples:

```text
Department = CSE
City = Chennai
Feedback = Good
Product Color = Blue
```

Qualitative data has two common types: **Nominal** and **Ordinal**.

Nominal Data

Nominal data consists of categories that **do not have a natural order or ranking**.

Examples:

```text
City:
Chennai
Mumbai
Delhi
```

There is no meaningful ranking such as Chennai > Mumbai > Delhi.

- Eye color
- Blood group
- Department
- Country
- Product category

> ***Nominal** = Names or categories without an order.*

Ordinal Data

Ordinal data consists of categories that **have a meaningful order or ranking**.

Examples:

```text
Poor → Average → Good → Excellent
```

There is a clear order from lower to higher satisfaction.

Other examples:

- Small → Medium → Large
- Beginner → Intermediate → Advanced
- Bronze → Silver → Gold

The important point is that the order matters, but the difference between categories is not necessarily equal.

For example, the difference between **Poor → Average** may not be the same as **Good → Excellent**.

Simple Comparison

Classification	Meaning	Example
Qualitative	Describes categories or qualities	Department = CSE
Nominal	Categories without order	City = Chennai
Ordinal	Categories with meaningful order	Rating = Good
Quantitative	Numerical quantity	Marks = 87
Discrete	Counted values	Students = 40
Continuous	Measured values	Height = 172.5 cm

Quick Memory Trick

```text
QUALITATIVE → QUALITY / CATEGORY

QUANTITATIVE → QUANTITY / NUMBER

NOMINAL → NO ORDER

ORDINAL → ORDER

DISCRETE → COUNT

CONTINUOUS → MEASURE
```

How This Relates to SQL

These classifications help us understand what values a database column represents.

For example:

Column	Example Value	Nature
Name	Arun	Qualitative
Department	CSE	Qualitative / Nominal
Feedback	Good	Qualitative / Ordinal
Student_Count	40	Quantitative / Discrete
Height	172.5	Quantitative / Continuous
Marks	87	Quantitative

SQL can work with both categorical and numerical data.

For example, we can filter by a category:

```sql
SELECT *
FROM Students
WHERE Department = 'CSE';
```

Or calculate values such as an average:

```sql
SELECT AVG(Marks) AS Average_Marks
FROM Students;
```

The first query works with a **categorical value**, while the second performs a numerical calculation.

> ***Data classification helps us understand what a value represents and how it can be analyzed.***

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
All numbers are quantitative	A number can sometimes act as a label or category; meaning matters
Discrete data must always be integers in every context	Discrete data represents separate countable values
Continuous data means every decimal number	Continuous data represents measurements that can vary across a range
Nominal categories have no useful information	They have useful category information; they simply have no natural ranking
Ordinal categories have equal intervals	Ordinal data provides order, but the gaps between categories are not necessarily equal
SQL can only work with quantitative data	SQL works with both categorical and numerical data

Placement Quick Points

```text
QUALITATIVE
→ CATEGORY / QUALITY
→ NOMINAL + ORDINAL

QUANTITATIVE
→ NUMBER / QUANTITY
→ DISCRETE + CONTINUOUS

NOMINAL
→ NO NATURAL ORDER

ORDINAL
→ MEANINGFUL ORDER

DISCRETE
→ COUNTED

CONTINUOUS
→ MEASURED
```

Copy

- **Qualitative data** describes categories or qualities.
- **Nominal data** has categories without a natural order.
- **Ordinal data** has categories with a meaningful order.
- **Quantitative data** represents numerical quantities.
- **Discrete data** is counted.
- **Continuous data** is measured and can take values within a range.
- SQL can filter, group, and analyze both categorical and numerical data.

Interview Questions

Q1. What is the difference between qualitative and quantitative data?

Qualitative data describes categories or qualities, while **quantitative data** represents numerical quantities that can be counted or measured.

Q2. What is nominal data?

Nominal data consists of categories with **no natural order**.

Example:

```
text City = Chennai, Mumbai, Delhi
```

Q3. What is ordinal data?

Ordinal data consists of categories with a **meaningful order or ranking**.

Example:

```
text Poor → Average → Good → Excellent
```

Q4. What is the difference between discrete and continuous data?

Discrete data is counted and consists of separate values, while **continuous data** is measured and can take values within a range.

Example:

```
text Discrete → Number of students = 40 Continuous → Height = 172.5 cm
```

Q5. Is customer feedback such as Poor, Average, Good, Excellent nominal or ordinal?

It is **ordinal** because the categories have a meaningful order.

Q6. Can qualitative data be used in SQL?

Yes. SQL can filter, group, sort, and count categorical data.

For example:

```sql
SELECT Department, COUNT(*) FROM Students GROUP BY Department;
```

Q7. Why is the distinction between discrete and continuous data useful?

It helps us understand whether a value represents something that is **counted** or **measured**, which affects how the data is interpreted and analyzed.

Practice & Hands-On Exercises

Beginner Practice

1. Define qualitative data in your own words.
2. Define quantitative data in your own words.
3. Give two examples of nominal data.
4. Give two examples of ordinal data.
5. Give two examples of discrete data.
6. Give two examples of continuous data.

Classify the Data

Classify each as **Nominal, Ordinal, Discrete, or Continuous**:

1. Number of employees in a company
2. Customer feedback: Poor, Average, Good, Excellent
3. Temperature of a room
4. Blood group
5. Number of orders placed

6. T-shirt size: S, M, L, XL
7. Student height
8. Department: CSE, IT, ECE

SQL Connection

Consider a `Students` table with:

```
TEXT

Name
Department
Marks
Height
```

1. Which column is categorical?
2. Which column contains a numerical score?
3. Which column can contain continuous measurements?
4. Write a query to find the number of students in each department:

```
TEXT

SELECT Department, COUNT(*) AS Total_Students
FROM Students
GROUP BY Department;
```

1. Write a query to calculate the average marks:

```
TEXT

SELECT AVG(Marks) AS Average_Marks
FROM Students;
```

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
DATA
│
├── QUALITATIVE
│   ├── NOMINAL → NO ORDER
│   └── ORDINAL → HAS ORDER
│
└── QUANTITATIVE
    ├── DISCRETE → COUNTED
    └── CONTINUOUS → MEASURED
```

Qualitative data describes categories, while quantitative data represents quantities. Nominal and ordinal describe the type of category; discrete and continuous describe the type of numerical value.

End of Classification of Data (Qualitative & Quantitative)

2.1 Multi-Tier Architecture

Level: 2 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Most modern applications are built by dividing the system into separate **tiers**, with each tier responsible for a specific job.

A common design is **Three-Tier Architecture**:

```
TEXT

CLIENT TIER
     ↓
APPLICATION TIER
     ↓
DATABASE TIER
```

Copy

This separation makes applications easier to develop, secure, maintain, and scale.

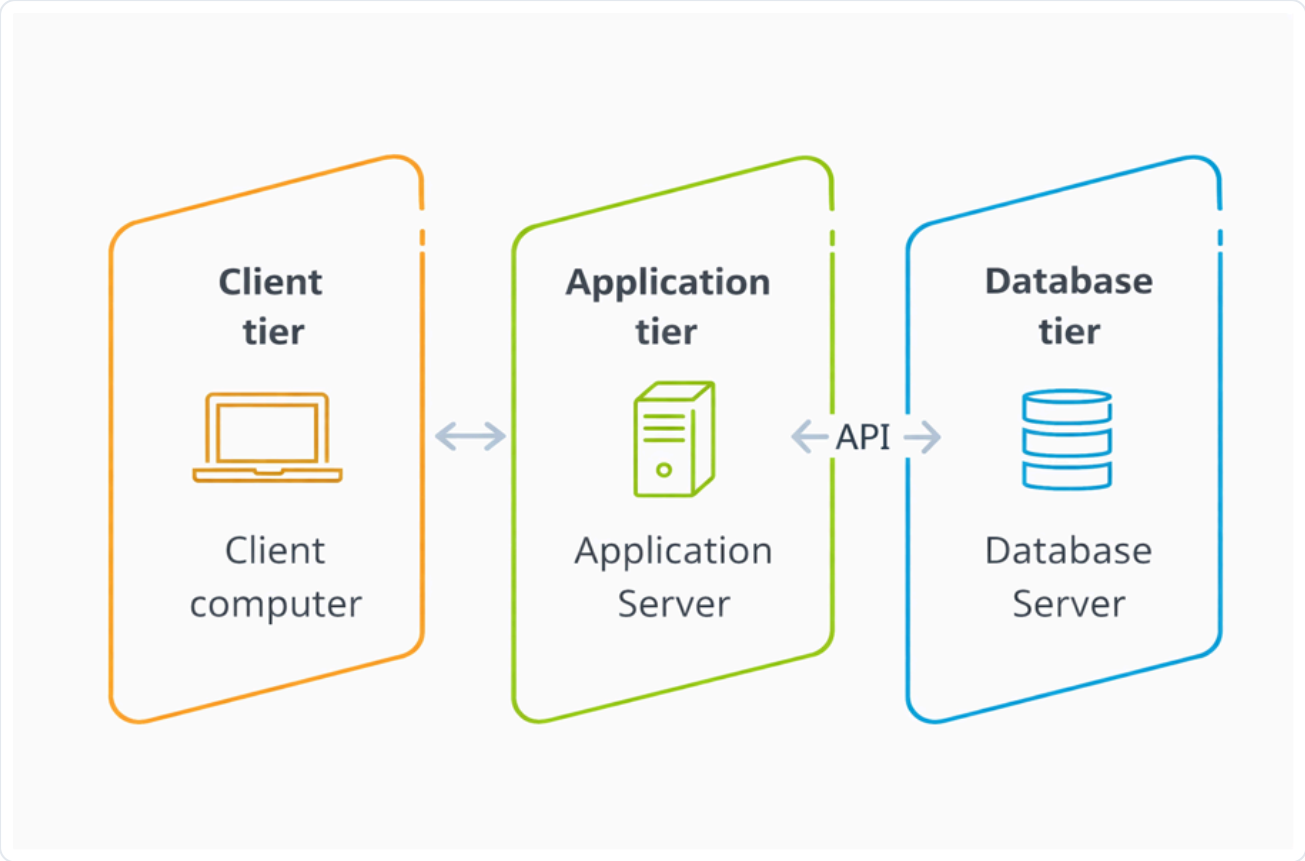


Figure 1: Three-tier architecture separates the user interface, application logic, and database into different tiers.

Client Tier

The **Client Tier** is the part of the application that the user sees and interacts with.

It is also called the **presentation layer**.

Examples include:

- Web browser
- Mobile application
- Desktop application

```
TEXT

USER
  ↓
CLIENT TIER
  ↓
Displays information
and collects user input
```

Copy

Application Tier

The **Application Tier** contains the application's processing and business logic.

It receives requests from the client, performs required processing, applies business rules, and communicates with the database when necessary.

For example, when a customer logs into an online banking application, the application tier can:

- Validate the login request
- Apply business rules
- Query customer information
- Prepare the response

```
TEXT

CLIENT REQUEST
       ↓
APPLICATION TIER
       ↓
BUSINESS LOGIC
       ↓
DATABASE REQUEST
```

Copy

Examples of technologies that can be used in this tier include:

- Python
- Java

- C#
- Node.js
- .NET

Database Tier

The **Database Tier** is responsible for storing and managing application data.

It typically contains the database server or database system.

Examples:

- MySQL
- PostgreSQL
- SQL Server
- Oracle

For example, an online banking database may store:

```text
Customers
Accounts
Transactions
Payments
```

The application tier sends database requests and receives the required results.

> ***The database tier stores and manages data; the application tier decides how that data is used by the application.***

How the Tiers Communicate

Consider an online shopping application.

When a user searches for a product:

```text
USER
 ↓
CLIENT
 ↓
APPLICATION
 ↓
DATABASE
 ↓
APPLICATION
 ↓
CLIENT
 ↓
USER
```

For example:

1. The user searches for **"Laptop"**.
2. The client sends the request to the application.
3. The application processes the request.
4. The application sends a database query.
5. The database returns matching records.
6. The application prepares the response.
7. The client displays the products.

The client normally **does not connect directly to the database**.

Important Note

The application tier communicates with the database using a **database connection, driver, or database protocol**. An API is normally used for communication between the client and application/service layer.

Why Use Three Tiers?

Separating the application into tiers provides several benefits.

Security

The database does not need to be directly exposed to users or public clients.

Maintainability

Changes to the user interface can often be made without changing database logic.

Scalability

Different tiers can be scaled independently when application traffic increases.

Business Logic Isolation

Business rules remain in the application tier instead of being placed entirely inside the client.

Connection Management

The application tier can manage database connections efficiently, including techniques such as connection pooling.

One-Tier, Two-Tier, and Three-Tier Architecture

The idea becomes clearer when we compare the common architectures.

1-Tier Architecture

All major components run together on the same machine.

```
TEXT

CLIENT
+
APPLICATION
+
DATABASE

       ↓

SAME MACHINE
```

Example:

> *A desktop application using a local SQLite database.*

2-Tier Architecture

The client communicates directly with the database server.

```
TEXT

CLIENT
   ↓
DATABASE SERVER
```

This is commonly called **client-server architecture**.

3-Tier Architecture

An application tier is placed between the client and database.

```text
CLIENT
   ↓
APPLICATION
   ↓
DATABASE
```

Copy

This is common in web applications.

Simple Comparison

Architecture	Basic Structure	Typical Example
1-Tier	Client + Logic + Database	Local desktop app with SQLite
2-Tier	Client → Database	Desktop client connected to database server
3-Tier	Client → Application → Database	Web or mobile application

Real-World Example: Online Banking

Consider an online banking application.

Client Tier

The customer uses:

```
TEXT

Web Browser / Mobile App
```

to enter:

```
TEXT

Username
Password
Transfer Amount
```

Application Tier

The application:

- Authenticates the user
- Validates the transfer
- Applies business rules
- Sends database requests

Database Tier

The database stores:

```
TEXT

Customer Details
Account Details
Transaction Details
```

The complete flow is:

```text
Customer
   ↓
Banking App
   ↓
Application Server
   ↓
Database
   ↓
Application Server
   ↓
Banking App
   ↓
Customer
```

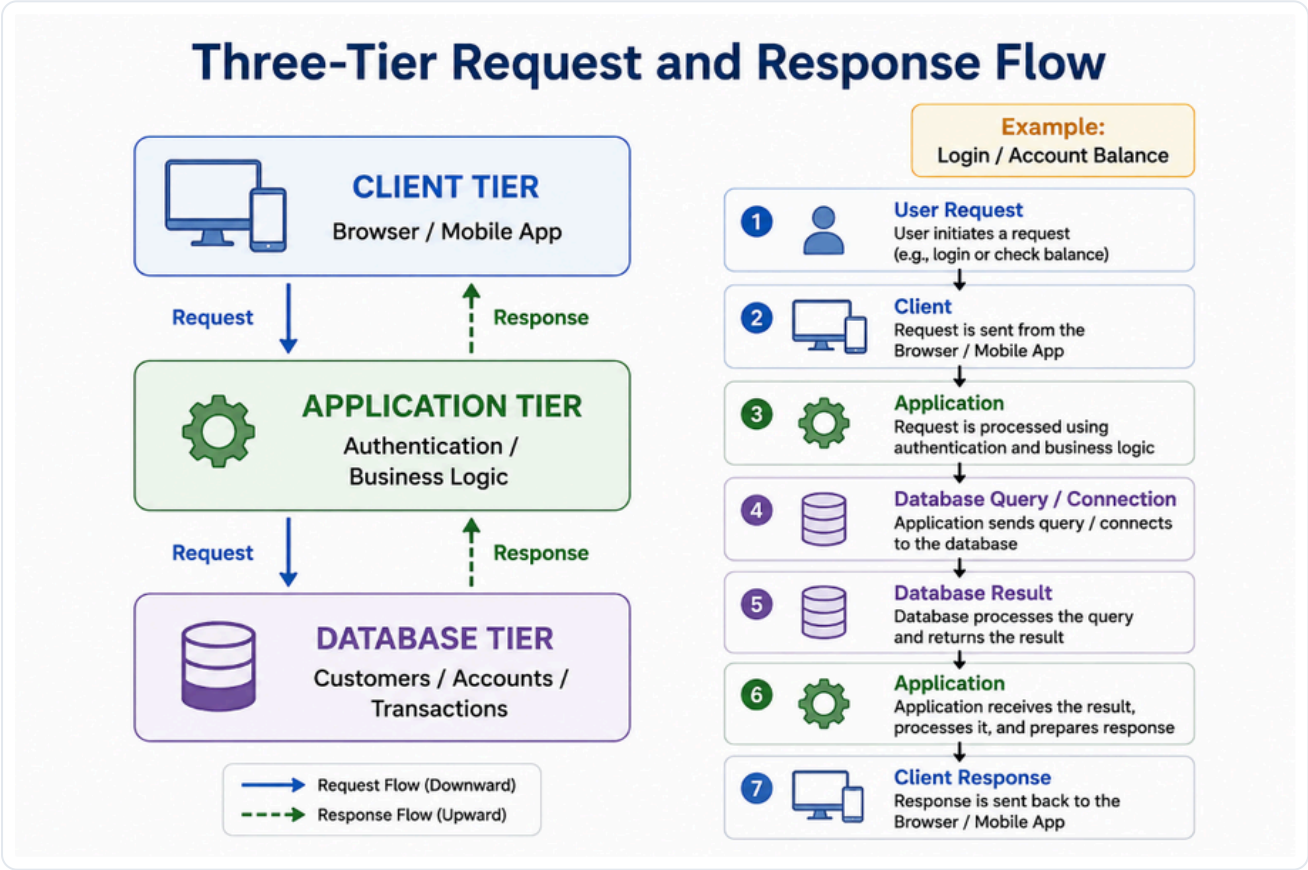


Figure 2: In a three-tier application, the client sends requests to the application tier, which communicates with the database and returns the result.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Client means only a physical computer	A client can be a browser, mobile app, or desktop application
The database is the application	The database stores and manages data; the application contains processing logic
The client should directly connect to the database in a web application	The application tier normally sits between the client and database
API always connects the application to the database	APIs commonly expose application/service functionality; database communication normally uses database drivers/connections/protocols
Three-tier means three physical computers	Tiers are logical layers; they may or may not run on separate physical or virtual machines
Three-tier architecture is required for every application	Simpler applications may use one-tier or two-tier designs

Placement Quick Points

```text
1-TIER
→ CLIENT + APPLICATION + DATABASE

2-TIER
→ CLIENT → DATABASE

3-TIER
→ CLIENT → APPLICATION → DATABASE
```

- **Client Tier** → user interface and user interaction.
- **Application Tier** → processing and business logic.
- **Database Tier** → data storage and management.
- Three-tier architecture separates responsibilities between layers.
- The client normally communicates with the application tier rather than directly with the database.
- The application tier communicates with the database through database connections, drivers, or protocols.
- Three-tier architecture can improve security, maintainability, and scalability.

Interview Questions

Q1. What is multi-tier architecture?

> Multi-tier architecture divides an application into separate logical layers, with each layer responsible for a specific function.

Q2. What are the three tiers in three-tier architecture?

They are:

```
text Client Tier Application Tier Database Tier
```

Q3. What is the role of the Client Tier?

It provides the user interface and allows users to interact with the application.

Q4. What is the role of the Application Tier?

It handles application processing, business rules, validation, and communication between the client and database.

Q5. What is the role of the Database Tier?

It stores and manages application data.

Q6. What is the difference between two-tier and three-tier architecture?

In two-tier architecture, the client communicates directly with the database.

In three-tier architecture, the application tier is placed between the client and database.

```text 2-TIER Client → Database

3-TIER Client → Application → Database ```

**Q7. Why is three-tier architecture preferred for many web applications?**

It separates the user interface, business logic, and database access. This can improve security, maintainability, and scalability.

**Q8. Give an example of a one-tier application.**

A local desktop application that stores its data in an embedded SQLite database on the same machine is an example of a one-tier design.
```

Q9. Does each tier need to run on a separate machine?

No. Tiers are logical layers. They can be deployed on the same machine or on separate servers depending on the application's requirements.

Practice & Hands-On Exercises

Beginner Practice

1. What is multi-tier architecture?
2. Name the three tiers in a three-tier architecture.
3. What does the Client Tier do?
4. What does the Application Tier do?
5. What does the Database Tier do?

Think and Answer

1. Explain the difference between 2-tier and 3-tier architecture.
2. Why should a web application usually avoid giving users direct database access?
3. What would happen if the application tier did not exist?
4. Does three-tier architecture always require three physical servers? Explain.

Real-World Practice

1. Draw the three-tier architecture of an online banking application.
2. Trace what happens when a user logs in.
3. Identify the tier responsible for each task:
4. Displaying a login form
5. Checking business rules
6. Storing customer details
7. Querying account information

Key Takeaway

```TEXT
USER
  ↓
CLIENT TIER
  ↓
APPLICATION TIER
  ↓
DATABASE TIER
  ↓
DATA
```

Copy

> *Three-tier architecture separates the user interface, application logic, and data management into different logical layers.*

End of Multi-Tier Architecture

2.2 What is a Database?

Level: 2 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **database** is an organized system used to **store, manage, retrieve, and update data**.

Think of a database like a well-organized digital storage room.

Instead of keeping information in separate files, a database organizes related data so that applications can find and work with it efficiently.

For example, a college database may store:

```text
Students
Courses
Teachers
Attendance
Marks
```

In a relational database, data is commonly organized into **tables**, which contain **rows and columns**.

```text
Database
   ↓
Tables
   ↓
Rows + Columns
   ↓
Data
```

A database is an organized system for storing and managing data so that it can be accessed and updated efficiently.

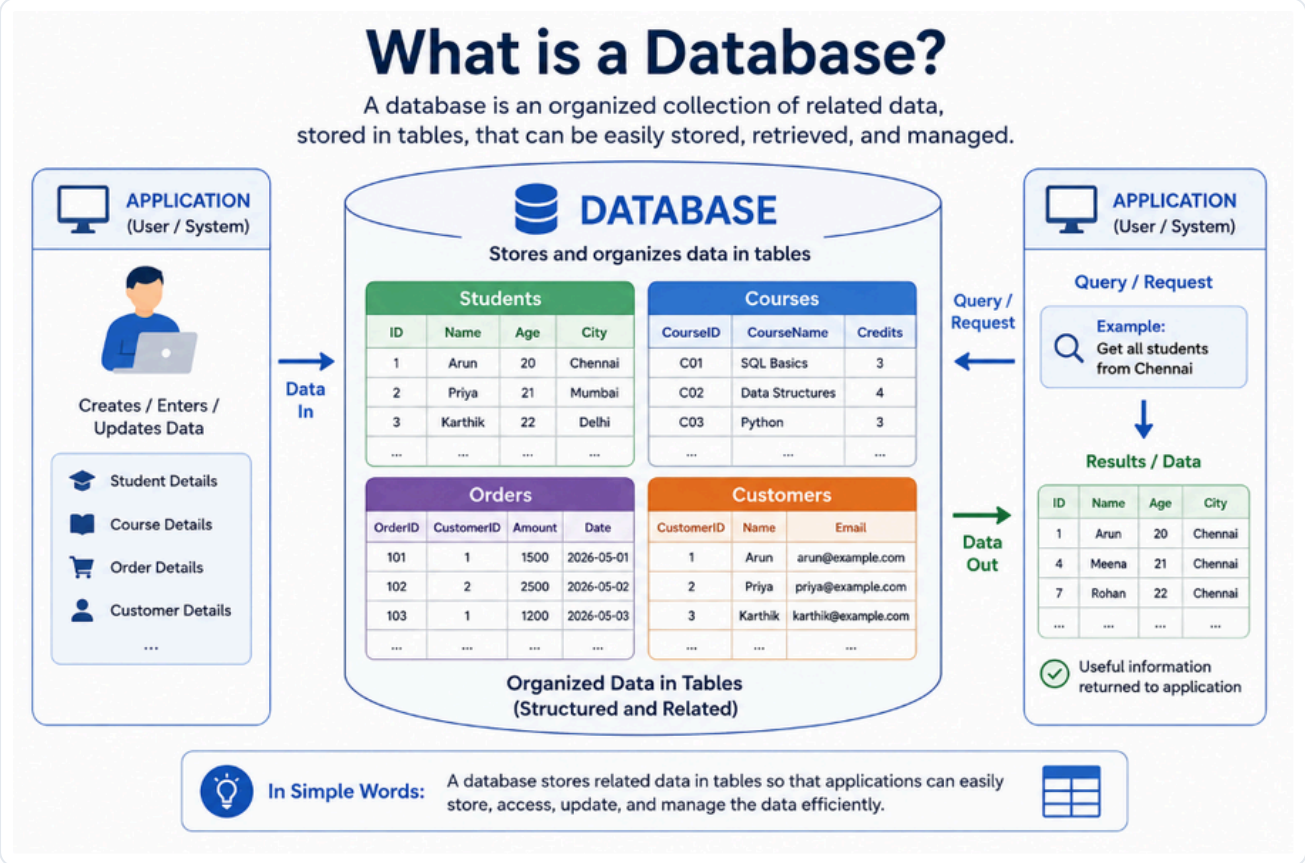


Figure 1: A database organizes related data into structures such as tables so applications can store, retrieve, and manage it efficiently.

Why Do We Need a Database?

Imagine a college storing information for **50,000 students**.

If every piece of information were kept in separate files, finding and updating records could become difficult.

- Which students belong to the CSE department?
- What marks did a particular student receive?
- Which students have low attendance?
- How many students are enrolled in a course?

A database makes these operations easier by organizing the data and providing tools to search and manage it.

Database vs Ordinary Files

A small application may store information in files such as:

```text
students.csv
students.xlsx
students.txt
```

Files can be useful for simple tasks, but large multi-user applications often need capabilities that ordinary files do not provide easily.

A database can provide:

- Structured organization
- Efficient searching
- Controlled multi-user access
- Data consistency
- Security and permissions
- Backup and recovery mechanisms

Simple Comparison

File-Based Storage	Database
Often managed as separate files	Centralized data management system
Searching can become difficult as data grows	Designed for efficient querying
Limited concurrency support	Supports controlled multi-user access
Relationships are harder to manage	Relationships can be represented directly
Security depends heavily on the application/file system	Database systems provide roles and permissions
Transaction support is limited or application-dependent	Database systems provide transaction mechanisms

> ***Files are suitable for many simple tasks, while databases are designed for reliable management of larger and shared datasets.***

What Is Stored Inside a Database?

In a relational database, related data is commonly divided into **tables**.

For example:

Students Table

Student_ID	Name	Department
101	Arun	CSE
102	Priya	IT
103	Karthik	CSE

Courses Table

Course_ID	Course_Name
C01	Python
C02	SQL
C03	Java

Here:

- A **database** can contain many tables.
- A **table** stores related data.
- A **row** represents one record.
- A **column** represents one attribute.

```text
DATABASE
   ↓
TABLE
   ↓
ROW + COLUMN
   ↓
DATA VALUE
```

This structure will become important when we learn SQL.

What Can We Do With a Database?

A database allows applications to perform common operations such as:

Store Data

Add new information.

Example:

```
TEXT
New student → Student_ID 104
```

Retrieve Data

Find required information.

Example:

```
TEXT
Find all CSE students
```

Update Data

Change existing information.

Example:

```
TEXT
Change a student's department
```

Copy

Delete Data

Remove information that is no longer required.

These four operations are commonly represented by **CRUD**:

```
TEXT
C → Create
R → Read
U → Update
D → Delete
```

Copy

SQL provides commands for performing these operations on relational databases.

How SQL Works With a Database

SQL is the language commonly used to communicate with relational databases.

Suppose a `Students` table contains:

Student_ID	Name	Department
101	Arun	CSE
102	Priya	IT
103	Karthik	CSE

We can use SQL to retrieve CSE students:

```sql
SELECT Name, Department
FROM Students
WHERE Department = 'CSE';
```

Result:

Name	Department
Arun	CSE
Karthik	CSE

The basic idea is:

```
APPLICATION
    ↓
SQL QUERY
    ↓
DATABASE
    ↓
RESULT
```

Data Persistence

One important feature of a database is **data persistence**.

Persistence means that stored data remains available even after the application stops or the computer/server is restarted, provided the data has been properly saved to non-volatile storage.

For example:

```text
Application Running
       ↓
Data Stored
       ↓
Application Stops
       ↓
Application Restarts
       ↓
Stored Data Still Available
```

This is different from temporary data that exists only in memory while a program is running.

Multi-User Access

Databases are commonly used by many users and applications at the same time.

For example, an online shopping system may simultaneously receive:

```text
Customer A → Places Order
Customer B → Checks Product
Customer C → Updates Address
Admin      → Views Sales Report
```

The database system manages these operations while protecting the consistency of the stored data.

This capability is especially important for systems such as:

- Banking
- E-commerce
- Hospital management
- College management
- Airline booking

Security and Access Control

A database can control **who is allowed to access or modify data**.

For example:

```
TEXT

Admin
→ Can read and modify data

Teacher
→ Can view student marks

Student
→ Can view their own records
```

Copy

Database systems can use users, roles, permissions, and privileges to control access.

> *Not every user should have permission to perform every operation.*

Why Databases Are Important in Real Applications

Consider an e-commerce application.

It may need to manage:

```text
Customers
Products
Orders
Payments
Inventory
Shipments
```

Thousands or millions of operations may happen over time.

A database provides a structured way to store these records and allows the application to efficiently retrieve, update, and manage them.

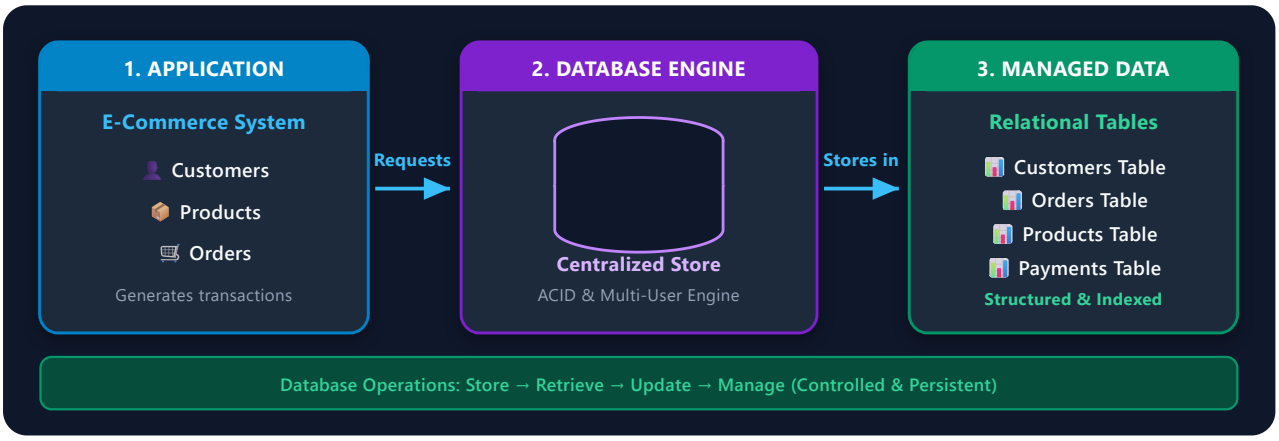


Figure 2: Applications use databases to store, retrieve, update, and manage large amounts of related data.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
A database is just a collection of files	A database is a system designed to organize and manage data
A database contains only one table	A database can contain many related tables
SQL and database are the same thing	A database stores and manages data; SQL is a language used to work with relational databases
Database means only storage	Databases also support querying, updates, transactions, security, concurrency, and recovery
Every database is relational	There are relational and non-relational database systems
A database is automatically secure	Security depends on configuration, access controls, authentication, and application design
More databases always mean better performance	Database design, queries, indexes, hardware, and workload all affect performance

Placement Quick Points

```text
DATABASE
→ ORGANIZED SYSTEM FOR STORING AND MANAGING DATA

RELATIONAL DATABASE
→ DATA ORGANIZED INTO TABLES

TABLE
→ ROWS + COLUMNS

CRUD
→ CREATE + READ + UPDATE + DELETE

SQL
→ USED TO WORK WITH RELATIONAL DATABASE DATA
```

Copy

- A database stores and manages data in an organized way.
- A relational database commonly stores data in tables containing rows and columns.
- Databases support data retrieval, insertion, updates, and deletion.
- Databases support multi-user access and access control.
- Persistence means stored data remains available after an application or system restarts.
- SQL is commonly used to communicate with and manipulate relational databases.
- Database performance depends on factors such as schema design, queries, indexes, hardware, and workload.

Interview Questions

Q1. What is a database?

> A database is an organized system used to store, manage, retrieve, and update data.

Q2. Why do we need databases?

Databases make it easier to organize, search, update, secure, and manage large amounts of data, especially when many users or applications need to access it.

Q3. What is the difference between a file system and a database?

A file system mainly manages individual files, while a database provides structured data management features such as querying, transactions, controlled multi-user access, and permissions.

Q4. What is data persistence?

Data persistence means that stored data remains available after an application stops or a system restarts, assuming the data has been properly saved to persistent storage.

Q5. Can a database contain multiple tables?

Yes. A relational database commonly contains multiple related tables.

For example:

```text
Students Courses Teachers Marks Attendance
```

Q6. What is CRUD?

CRUD represents four common data operations:

```text
C → Create R → Read U → Update D → Delete
```

Q7. What is the relationship between SQL and a database?

A database stores and manages data, while SQL is a language commonly used to retrieve and manipulate data in relational databases.

Q8. What is concurrency in a database?

Concurrency is the ability of a database system to handle multiple users or transactions accessing data at the same time while maintaining data consistency.

Practice & Hands-On Exercises

Beginner Practice

1. Define a database in your own words.
2. Why is a database needed when an application has a large amount of data?
3. What is the difference between a database and a file?
4. What is a table?
5. What are rows and columns?

Think and Answer

1. Why would a college with thousands of students benefit from using a database?
2. Explain data persistence with a real-world example.
3. Why is multi-user access important in an online shopping application?
4. Why should different users have different database permissions?

SQL Connection

Consider this table:

Student_ID	Name	Department
101	Arun	CSE
102	Priya	IT
103	Karthik	CSE

1. Write a query to display all students:

```text
SELECT *
FROM Students;
```

1. Write a query to display only CSE students:

```text
SELECT Name, Department
FROM Students
WHERE Department = 'CSE';
```

1. Which part of the table represents a record?
2. Which part represents an attribute?
3. Identify the database operation represented by adding a new student.
4. Identify the database operation represented by changing a student's department.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```sql
DATABASE
   ↓
ORGANIZES DATA
   ↓
STORES + RETRIEVES + UPDATES
   ↓
APPLICATIONS USE THE DATA
   ↓
SQL HELPS WORK WITH RELATIONAL DATA
```

> ***A** database is an organized system for storing and managing data so applications can reliably access, update, and use that data.*

End of What is a Database?

2.3 DBMS (Database Management System)

Level: 2 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

DBMS stands for **Database Management System**.

A DBMS is software that helps us **create, store, retrieve, update, and manage data in a database**.

Think of it as a **manager between an application and the database**.

For example:

```
TEXT

USER / APPLICATION
        ↓
       DBMS
        ↓
    DATABASE
```

The DBMS controls how data is stored and how users and applications access it.

What Does a DBMS Do?

A DBMS helps us to:

- Store data
- Retrieve data
- Insert new data
- Update existing data
- Delete data
- Control user access

- Manage multiple users
- Protect data integrity
- Recover data after failures

> *A DBMS is software used to create, manage, and control access to databases.*

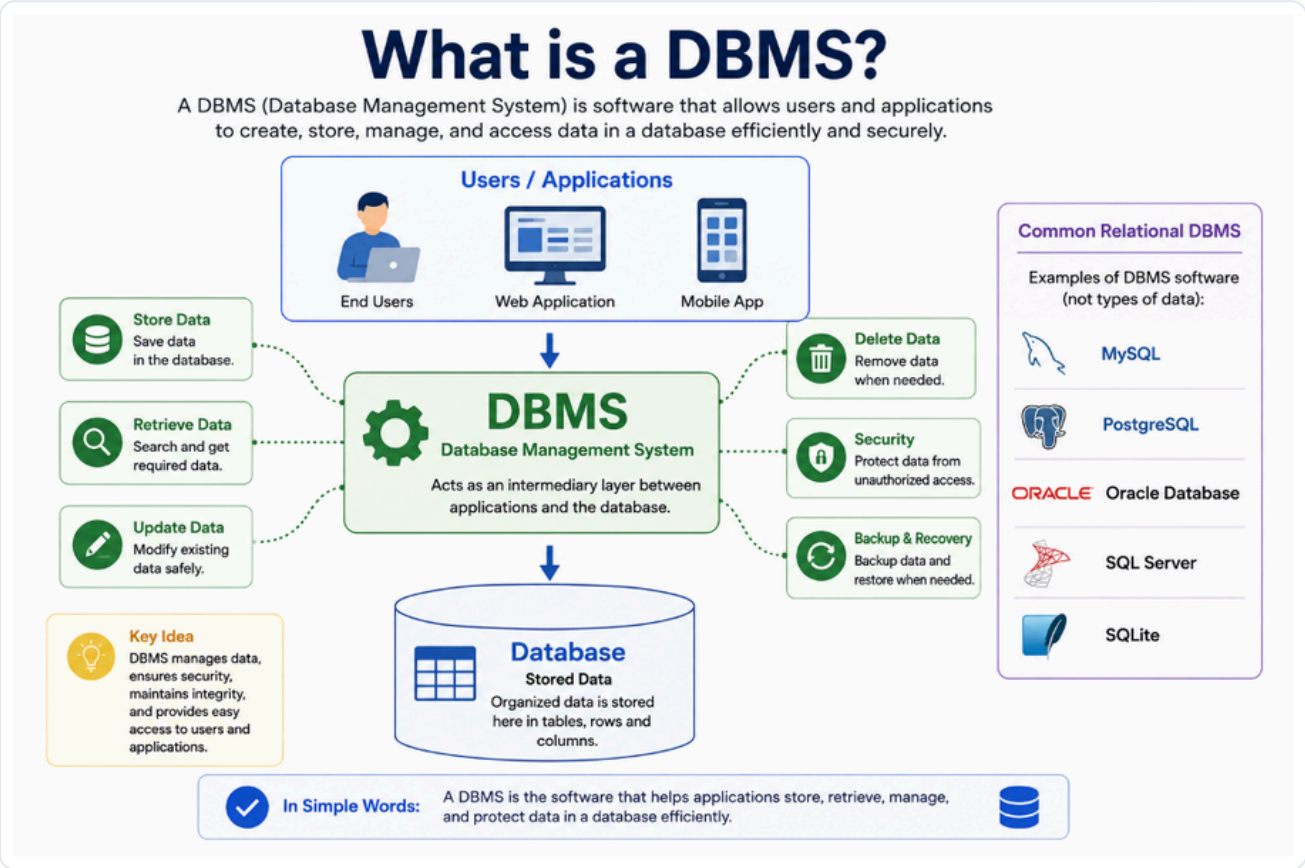


Figure 1: A DBMS acts as the software layer that manages databases and controls how applications access data.

Database vs DBMS

Students often confuse **database** and **DBMS**.

Database

A **database** is the organized collection of data.

For example:

```text
Students
Courses
Orders
Customers
```

DBMS

A **DBMS** is the software used to manage that database.

Examples:

```text
PostgreSQL
MySQL
Oracle Database
SQL Server
SQLite
```

A simple analogy:

> *Database = the data being stored*
> *DBMS = the software that manages that data*

How a DBMS Works

Suppose a student application wants to find all students from the CSE department.

The application sends a request through the DBMS:

```text
Application
    ↓
   DBMS
    ↓
Database
    ↓
  Result
    ↓
   DBMS
    ↓
Application
```

For example, the application may send:

```sql
SELECT *
FROM Students
WHERE Department = 'CSE';
```

The DBMS receives the SQL statement, processes it, obtains the required data from the database, and returns the result to the application.

> *The DBMS is responsible for processing database requests and managing access to the stored data.*

Main Responsibilities of a DBMS

1. Data Storage Management

The DBMS manages how database data is stored and accessed.

2. Data Retrieval

It processes queries and returns the required data.

3. Data Modification

It allows applications to insert, update, and delete data.

4. Security

It can control who is allowed to access specific databases or operations.

5. Multi-User Access

Multiple users or applications can work with the database while the DBMS manages their access.

6. Backup and Recovery

A DBMS provides mechanisms that help protect data and recover from certain failures.

7. Data Integrity

The DBMS can enforce rules that help keep data correct and consistent.

Common DBMS Examples

Some commonly used database management systems include:

DBMS	Common Use
MySQL	Web applications and business applications
PostgreSQL	Web applications, analytics, and enterprise systems
Oracle Database	Large enterprise and mission-critical systems
Microsoft SQL Server	Enterprise applications and Microsoft-based environments
SQLite	Embedded and lightweight applications

Note: MongoDB is also a database management system, but it is a **document-oriented NoSQL DBMS**, rather than a relational DBMS.

DBMS and SQL

SQL and **DBMS** are also different concepts.

SQL

SQL is a **language** used to communicate with relational databases.

Example:

```sql
SELECT Name
FROM Students;
```

DBMS

The DBMS is the **software system** that receives and processes SQL commands and manages the database.

```text
SQL
 ↓
DBMS
 ↓
DATABASE
 ↓
RESULT
```

For example, you can write SQL in PostgreSQL, MySQL, or SQL Server, but each DBMS has its own implementation details and additional features.

Why Do We Need a DBMS?

Imagine an application storing thousands or millions of customer records.

The application needs more than simple file storage.

- Find data quickly
- Allow many users to work at the same time
- Protect sensitive data
- Prevent invalid data
- Handle transactions
- Recover from failures

A DBMS provides the software capabilities needed to manage these requirements.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Database and DBMS are the same	A database contains data; a DBMS is software that manages the database
SQL is a DBMS	SQL is a language; MySQL, PostgreSQL, Oracle Database, and SQL Server are DBMS products
A DBMS only stores data	It also retrieves, updates, secures, and manages data
Every DBMS is relational	DBMS products can use different database models, including relational and document-oriented models
DBMS and database server always mean separate machines	A DBMS is software; its deployment can vary depending on the architecture
SQL works directly with the disk	The DBMS processes SQL requests and manages access to stored data

Placement Quick Points

```text
DATABASE
→ ORGANIZED DATA

DBMS
→ SOFTWARE THAT MANAGES THE DATABASE

SQL
→ LANGUAGE USED TO COMMUNICATE WITH RELATIONAL DATABASES
```

- DBMS stands for **Database Management System**.
- A DBMS is software used to create, manage, and access databases.
- It handles operations such as storing, retrieving, updating, and deleting data.
- A DBMS can provide security, concurrency, integrity, backup, and recovery features.
- MySQL, PostgreSQL, Oracle Database, SQL Server, and SQLite are common relational DBMS examples.
- MongoDB is a document-oriented NoSQL DBMS.
- SQL is a language; a DBMS is the software that processes SQL and manages database resources.

Interview Questions

Q1. What is a DBMS?

> A DBMS is software used to create, store, manage, retrieve, and control access to data in a database.

Q2. What is the difference between a database and a DBMS?

A **database** is the organized collection of data, while a **DBMS** is the software used to manage that data.

Q3. What are the main responsibilities of a DBMS?

A DBMS manages data storage, retrieval, modification, security, multi-user access, integrity, backup, recovery, and transaction processing.

Q4. What is the difference between SQL and DBMS?

SQL is a language used to communicate with relational databases. A DBMS is the software that processes SQL statements and manages the underlying database.

Q5. Name some common relational DBMS.

Examples include:

```text
MySQL PostgreSQL Oracle Database Microsoft SQL Server SQLite
```

Q6. Is MongoDB a DBMS?

Yes. MongoDB is a **document-oriented NoSQL DBMS**, not a relational DBMS.

Q7. Why is a DBMS important for multi-user applications?

It manages simultaneous access to shared data and provides mechanisms for maintaining consistency, security, and reliable transactions.

Q8. What happens when an application sends an SQL query?

The DBMS receives and processes the SQL statement, accesses the required database data, and returns the result to the application.

Practice & Hands-On Exercises

Beginner Practice

1. What does DBMS stand for?
2. Define DBMS in your own words.
3. What is the difference between a database and a DBMS?
4. List five common DBMS products.
5. List four responsibilities of a DBMS.

Think and Answer

1. Why can't a large application simply store all its data in separate text files?
2. Why is security important in a DBMS?
3. Why is multi-user access important?
4. What is the difference between SQL and a DBMS?
5. Why is MongoDB different from PostgreSQL?

SQL Connection

Consider:

```text
SELECT Name
FROM Students
WHERE Department = 'CSE';
```

Copy

1. What is SQL doing in this example?
2. What component receives and processes this SQL query?
3. Where is the actual student data stored?
4. What does the application receive after the query is processed?

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```SQL
USER / APPLICATION
        ↓
       SQL
        ↓
      DBMS
        ↓
    DATABASE
        ↓
     RESULT
```

A DBMS is the software layer that manages databases and provides a controlled way for applications and users to store, retrieve, and manage data.

End of DBMS (Database Management System)

2.4 RDBMS and the Role of SQL

Level: 2 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

RDBMS stands for **Relational Database Management System**.

An RDBMS is a type of **DBMS** that stores data mainly in **tables**, using rows and columns. Related tables can be connected using keys.

For example, a college system may have:

```text
Students
Courses
Departments
Marks
```

A relationship can connect these tables. For example, a student can belong to a department, and a student can have many marks records.

> ***RDBMS is a type of DBMS that organizes data into related tables.***

SQL stands for **Structured Query Language**. It is the language commonly used to interact with relational databases.

```text
RDBMS → Stores and manages relational data
SQL   → Used to query and manipulate that data
```

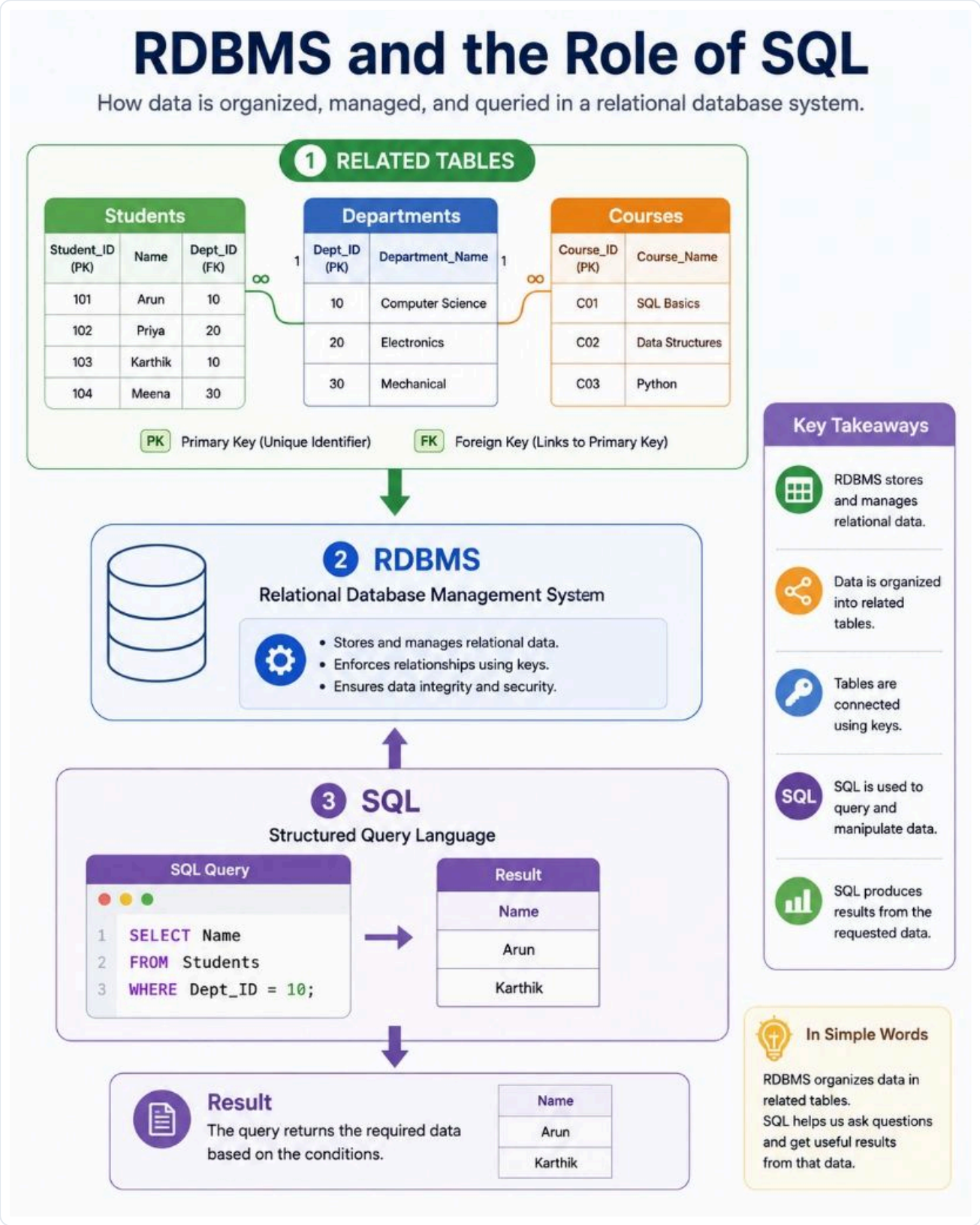


Figure 1: An RDBMS stores related data in tables, while SQL is used to query and manipulate that data.

What Makes an RDBMS Relational?

The word **relational** refers to the relationships between tables.

Consider two tables:

Departments

Dept_ID	Department_Name
10	CSE
20	IT

Students

Student_ID	Name	Dept_ID
101	Arun	10
102	Priya	20
103	Karthik	10

Here, `Dept_ID` in the `Students` table refers to `Dept_ID` in the `Departments` table.

This allows the database to represent the relationship:

```text
Departments
     ↑
     |
   Dept_ID
     |
     ↓
Students
```

The database can use keys to define and maintain these relationships.

What Is a Table in an RDBMS?

A table is a structured collection of related data.

For example:

Student_ID	Name	Department	Marks
101	Arun	CSE	87
102	Priya	IT	91
103	Karthik	CSE	78

In a relational table:

- **Row** → one record
- **Column** → one attribute
- **Table** → collection of related records

Tables provide a consistent structure for storing and querying data.

Role of SQL

SQL is the language used to communicate with relational database systems.

We can use SQL to:

- Retrieve data
- Insert data
- Update data
- Delete data
- Create database objects
- Define relationships and constraints
- Aggregate and analyze data

For example:

```sql
SELECT Name, Marks
FROM Students
WHERE Marks > 80;
```

The database returns:

Name	Marks
Arun	87
Priya	91

The SQL statement tells the database **what result is required**.

How SQL and an RDBMS Work Together

A simple request flow is:

```text
USER / APPLICATION
        ↓
    SQL QUERY
        ↓
      RDBMS
        ↓
   TABLE DATA
        ↓
    RESULT
```

For example, a student management application may send:

```sql
SELECT *
FROM Students
WHERE Department = 'CSE';
```

The RDBMS processes the query and returns the matching records.

Important Idea

SQL is **not the database**.

SQL is the language used to communicate with the database.

The RDBMS is the software that manages the relational database and processes SQL statements.

Declarative Nature of SQL

SQL is commonly described as a **declarative language**.

This means you generally tell the database **what data you want**, rather than specifying every step required to retrieve it.

For example:

```sql
SELECT *
FROM Employees
WHERE Salary > 50000;
```

Copy

You specify:

> *"Give me employees whose salary is greater than 50,000."*

You do not normally tell the database:

> *"Scan this file, compare each row, store matching rows in memory, then sort them."*

The RDBMS's query optimizer determines an efficient execution strategy.

Simple Idea

```text
SQL
  ↓
WHAT DATA IS NEEDED?
  ↓
RDBMS QUERY PROCESSING
  ↓
HOW TO GET IT?
  ↓
RESULT
```

> *SQL describes the required result; the database system determines how to execute the request.*

Real-World Example: Online Shopping

Consider an online shopping application.

The RDBMS might contain:

```text
Customers
Products
Orders
Order_Items
Payments
```

When a customer searches for products, the application sends an SQL query.

```text
Customer
   ↓
Shopping App
   ↓
SQL Query
   ↓
RDBMS
   ↓
Products Table
   ↓
Search Result
```

The application then displays the result to the customer.

This is how SQL becomes the communication language between an application and a relational database system.

RDBMS vs DBMS

An RDBMS is a **specific type of DBMS**.

DBMS	RDBMS
General term for database management software	Type of DBMS based on the relational model
Can support different database models	Organizes data into related tables
May or may not use relationships between tables	Uses relationships between tables
Examples depend on the database model	PostgreSQL, MySQL, Oracle Database, SQL Server

> *Every RDBMS is a DBMS, but not every DBMS is an RDBMS.*

Examples of RDBMS

Common relational database management systems include:

- PostgreSQL
- MySQL
- Oracle Database
- Microsoft SQL Server
- SQLite

They support relational tables and SQL, although each product has its own features, syntax extensions, and implementation details.

Brief History of the Relational Model

The **relational model** was introduced by **Edgar F. Codd at IBM in 1970**.

The model provided a mathematical foundation for representing data as relations, which are commonly implemented as tables in relational database systems.

For beginners, the main idea to remember is:

> *The relational model organizes data into related structures, and modern RDBMS products implement this model using tables, keys, constraints, and SQL.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
RDBMS and DBMS are completely different things	RDBMS is a type of DBMS
SQL is an RDBMS	SQL is a language used to work with an RDBMS
Every table must be unrelated to other tables	Tables can be connected through keys and relationships
SQL tells the database exactly how to access the disk	SQL generally specifies what result is required; the optimizer chooses an execution plan
A foreign key stores the entire related table	A foreign key stores a value that refers to a key in another table
Every DBMS is relational	Database systems can use different models; RDBMS specifically uses the relational model

Placement Quick Points

```text
RDBMS
→ RELATIONAL DATABASE MANAGEMENT SYSTEM
→ STORES DATA IN RELATED TABLES

TABLE
→ ROWS + COLUMNS

KEYS
→ HELP IDENTIFY RECORDS AND CONNECT TABLES

SQL
→ LANGUAGE USED TO QUERY AND MANIPULATE RELATIONAL DATA

SQL
→ WHAT DATA IS NEEDED

RDBMS
→ HOW TO EXECUTE THE REQUEST
```

- RDBMS is a type of DBMS based on the relational model.
- Relational data is commonly organized into tables.
- Tables can be related using keys such as primary keys and foreign keys.
- SQL is the standard language commonly used to interact with relational databases.
- SQL is declarative: it describes the desired result, while the database system determines an execution strategy.
- PostgreSQL, MySQL, Oracle Database, SQL Server, and SQLite are common RDBMS products.
- Edgar F. Codd introduced the relational model in 1970.

Interview Questions

Q1. What is an RDBMS?

RDBMS stands for **Relational Database Management System**. It is a type of DBMS that stores and manages data using related tables.

Q2. What is the difference between DBMS and RDBMS?

A DBMS is a general term for software used to manage databases. An RDBMS is a type of DBMS that follows the relational model and organizes data into related tables.

Q3. What is the role of SQL in an RDBMS?

SQL is used to retrieve, insert, update, delete, define, and analyze data in a relational database.

Q4. Why is SQL called a declarative language?

Because the user specifies **what result is required**, while the database system decides **how to execute the request**.

Q5. Who introduced the relational model?

Edgar F. Codd introduced the relational model at IBM in **1970**.

Q6. Give examples of RDBMS.

Examples include:

```text
PostgreSQL
MySQL
Oracle Database
Microsoft SQL Server
SQLite
```

Q7. How are tables related in an RDBMS?

Tables can be related using keys. A common example is a **foreign key** referencing a key in another table.

Q8. What happens when an application sends an SQL query?

The RDBMS receives the SQL statement, processes it, accesses the required data, and returns the result to the application.

Practice & Hands-On Exercises

Beginner Practice

1. What does RDBMS stand for?
2. Define RDBMS in your own words.
3. What is the difference between DBMS and RDBMS?
4. What is the role of SQL?
5. Name five RDBMS products.

Think and Answer

1. Why are relationships between tables useful?
2. Why is SQL called declarative?
3. What is the difference between a table, a row, and a column?
4. Why should an application normally use the RDBMS instead of directly accessing database files?

SQL Connection

Consider:

```
TEXT

Departments
Dept_ID | Department_Name

Students
Student_ID | Name | Dept_ID
```

1. Which column connects the two tables?
2. Explain why `Students` is related to `Departments`.
3. Write a SQL query to display students belonging to the `CSE` department:

```text
SELECT s.Student_ID, s.Name, d.Department_Name
FROM Students s
JOIN Departments d ON s.Dept_ID = d.Dept_ID
WHERE d.Department_Name = 'CSE';
```

1. Write a SQL query to display all employees from the `IT` department:

```sql
SELECT *
FROM Employees
WHERE Department = 'IT';
```

1. Explain what part of the SQL statement represents **what data is needed**.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```sql
APPLICATION
     ↓
    SQL
     ↓
   RDBMS
     ↓
RELATED TABLES
     ↓
    RESULT
```

> ***RDBMS stores data in related tables, while SQL provides the language used to query and manipulate that relational data.***

End of RDBMS and the Role of SQL

2.5 Core Database Characteristics

Level: 2 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A good database system should do more than simply store data.

It should keep data **organized, accurate, secure, available, and usable by multiple users**.

The core characteristics of a database include:

1. Reduced Data Redundancy
2. Large-Scale Data Management
3. Data Integrity
4. Data Security
5. Data Consistency
6. Backup and Recovery
7. Privacy
8. Data Sharing

A simple way to remember the goal is:

```text
STORE DATA
    ↓
KEEP IT CORRECT
    ↓
KEEP IT SECURE
    ↓
MAKE IT AVAILABLE
    ↓
LET AUTHORIZED USERS USE IT
```

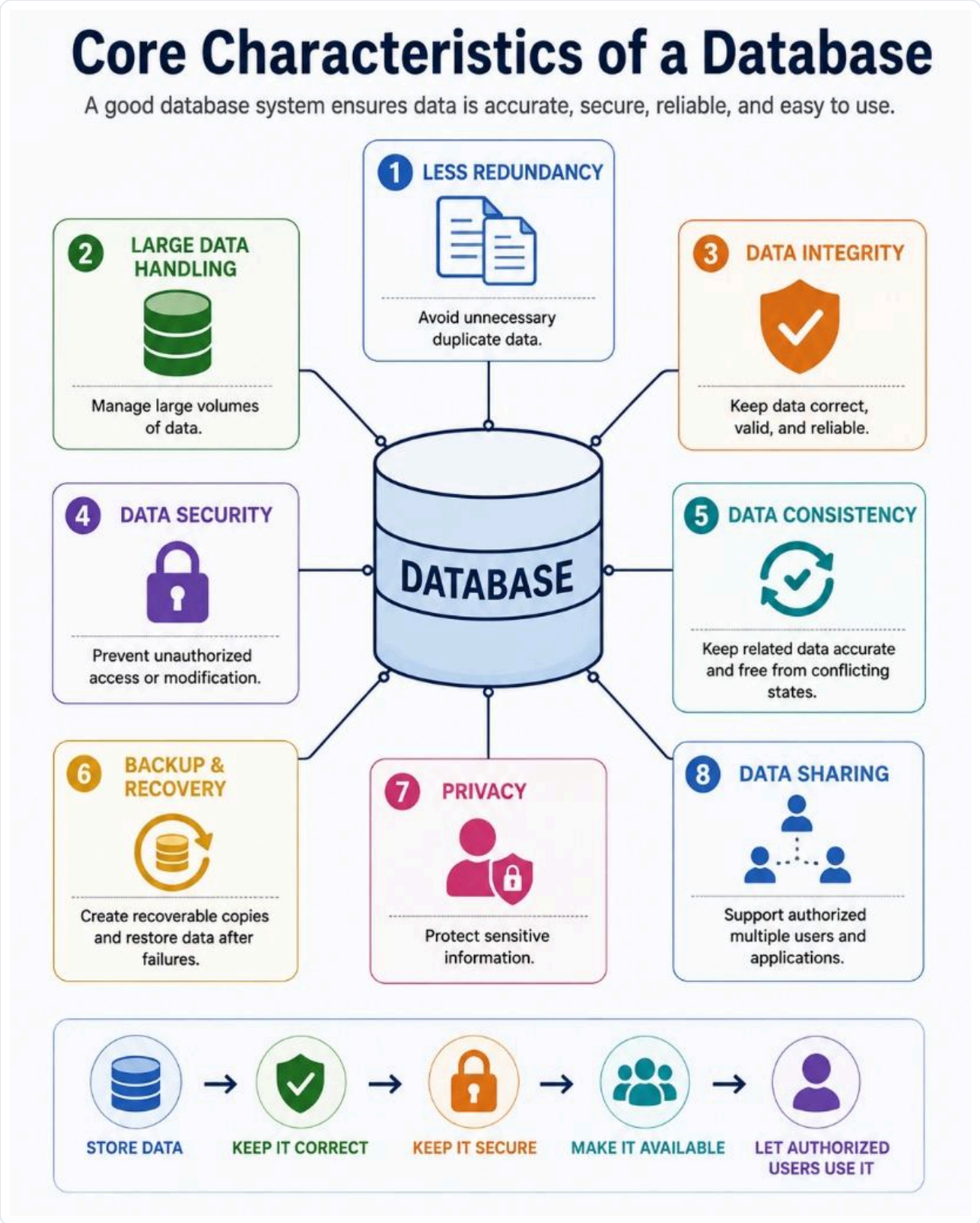


Figure 1: A good database system keeps data organized, accurate, secure, recoverable, and accessible to authorized users.

1. Reduced Data Redundancy

Data redundancy means storing the same piece of data unnecessarily in multiple places.

For example, suppose a customer's phone number is stored repeatedly in several files:

```text
Customer File    → 9876543210
Order File       → 9876543210
Payment File     → 9876543210
```

If the customer changes the number, every copy may need to be updated.

A well-designed database reduces unnecessary duplication.

> ***Reduced redundancy means avoiding unnecessary copies of the same data.***

2. Managing Large Amounts of Data

Database systems are designed to store and manage large volumes of data.

For example, an e-commerce application may store:

```text
Customers
Products
Orders
Payments
Shipments
```

The database should allow applications to search and manage this data efficiently as the amount of data grows.

3. Data Integrity

Data integrity means keeping data **correct, valid, and reliable**.

For example, if a student's age must be a positive number, the database should prevent invalid values such as:

```
TEXT
Age = -20
```

Copy

Relational databases can use rules such as:

- `PRIMARY KEY`
- `FOREIGN KEY`
- `NOT NULL`
- `UNIQUE`
- `CHECK`

to help enforce data integrity.

> *Integrity asks: "Is the data valid and correct?"*

4. Data Security

Data security protects data from unauthorized access or modification.

For example:

```text
Student
→ Can view their own records

Teacher
→ Can view authorized student information

Admin
→ Can manage required records
```

Database systems can use authentication, roles, privileges, and permissions to control access.

> *Security asks: "Who is allowed to access or change the data?"*

5. Data Consistency

Data consistency means data remains accurate and does not contain conflicting values when it is accessed or updated.

For example, imagine an order system showing:

```text
Order System      → Payment = Paid
Payment System    → Payment = Pending
```

Such a conflict can cause problems.

Database transactions and integrity rules help maintain consistent data.

Integrity vs Consistency

These terms are related but slightly different:

Integrity	Consistency
Data follows defined rules	Data does not conflict across related operations
Focuses on validity	Focuses on correct state
Example: Age cannot be negative	Example: completed payment should not remain incorrectly marked as pending

6. Backup and Recovery

Data can be lost because of:

- Hardware failure
- Software errors
- Accidental deletion
- System crashes
- Other failures

A database system can support **backup and recovery** mechanisms so data can be restored after certain failures.

For example:

```text
DATABASE
   ↓
BACKUP
   ↓
SYSTEM FAILURE
   ↓
RECOVERY
   ↓
DATA RESTORED
```

Copy

> *Backup protects a copy of data; recovery restores data after a failure.*

---

7. Privacy

Privacy means protecting sensitive information and ensuring that only authorized people can access it.

Examples of sensitive data include:

- Personal details
- Salary information
- Banking information
- Student records
- Health records

A database system can support privacy through permissions, access controls, encryption, and appropriate application design.

---

8. Data Sharing

A database often needs to support multiple authorized users and applications at the same time.

For example, in an online shopping system:

```text
Customer  → Places Order
Warehouse → Updates Inventory
Payment   → Confirms Payment
Admin     → Checks Order
```

All of these systems may work with shared database data.

The DBMS manages concurrent access so that users and transactions can work safely.

Additional Database Characteristics

The characteristics above describe what a database system should provide. Database systems also have several important concepts that help achieve these goals.

Self-Describing Nature

A database system stores **metadata**, which means **data about data**.

For example, metadata can describe:

```text
Table Name
Column Name
Data Type
Constraints
Indexes
Permissions
```

This information is typically maintained in a **system catalog** or data dictionary.

Example

Suppose a table contains:

```text
Student_ID → INTEGER
Name       → VARCHAR
Marks      → INTEGER
```

The actual values are data.

The information describing those columns is **metadata**.

> *Data = the stored values. Metadata = information describing the stored data.*

Program-Data Independence

Program-data independence means application programs can remain largely unchanged when certain aspects of how data is stored or organized are changed.

For example, an administrator might change the physical storage structure without requiring the application to be rewritten.

This separation makes database systems easier to maintain.

```
TEXT

APPLICATION
     ↓
DATABASE SYSTEM
     ↓
PHYSICAL STORAGE
```

The application does not need to know every detail of the physical storage.

Multiple Views of Data

Different users may need different parts of the same database.

For example:

```
TEXT

Student
→ Name, Course, Marks

Teacher
→ Student Name, Marks, Attendance

Admin
→ Student, Course, Fee, Attendance
```

A **view** can provide a customized representation of database data.

For example:

```
SQL

CREATE VIEW Student_Public_Info AS
SELECT Student_ID, Name, Department
FROM Students;
```

A view can expose only the information a particular user or application needs.

Multi-User Transaction Processing

A database can have many users performing operations at the same time.

For example:

```
TEXT
Copy

Customer A → Books Seat 10
Customer B → Books Seat 11
Customer C → Checks Available Seats
```

The DBMS manages these operations using mechanisms such as **transactions, locking, and isolation**.

The goal is to prevent concurrent operations from producing incorrect or conflicting results.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Redundancy and inconsistency are the same	Redundancy is unnecessary duplication; inconsistency means conflicting or incorrect values
Security means the database is automatically safe	Security requires proper authentication, permissions, configuration, and application controls
Backup and recovery are the same thing	Backup creates a recoverable copy; recovery restores data after a failure
Privacy and security are exactly the same	Security provides controls; privacy concerns appropriate protection and use of personal information
Data integrity means data can never change	Integrity means data remains valid and follows defined rules
A view stores a completely separate copy of the data	A traditional database view is a virtual representation based on a query
Multiple users can freely modify the same data without control	DBMS concurrency mechanisms help coordinate simultaneous operations

Placement Quick Points

```text
REDUNDANCY
→ AVOID UNNECESSARY DUPLICATION

INTEGRITY
→ KEEP DATA VALID AND CORRECT

CONSISTENCY
→ AVOID CONFLICTING DATA STATES

SECURITY
→ CONTROL ACCESS

PRIVACY
→ PROTECT SENSITIVE INFORMATION

BACKUP
→ CREATE A RECOVERABLE COPY

RECOVERY
→ RESTORE DATA AFTER FAILURE

SHARING
→ SUPPORT AUTHORIZED MULTIPLE USERS
```

- Databases reduce unnecessary data duplication.
- Data integrity ensures values follow defined rules.
- Data consistency helps prevent conflicting data states.
- Security controls who can access or modify data.
- Privacy focuses on protecting sensitive information.
- Backup and recovery help protect against data loss.
- Databases support controlled sharing among multiple users.
- Metadata describes the structure and properties of stored data.
- Views can provide different users with customized representations of data.
- Transactions and concurrency controls help manage simultaneous database operations.

Interview Questions

Q1. What is data redundancy?

Data redundancy is the unnecessary storage of the same data in multiple places.

Q2. What is data integrity?

Data integrity means that data remains accurate, valid, and follows defined rules.

Q3. What is the difference between data integrity and data consistency?

Integrity focuses on whether data satisfies defined validity rules. **Consistency** focuses on whether the database remains in a correct, non-conflicting state.

Q4. What is data security?

Data security is the protection of data from unauthorized access, modification, or misuse.

Q5. What is the difference between backup and recovery?

A **backup** is a stored copy of data that can be used later. **Recovery** is the process of restoring data after a failure or loss.

Q6. What is metadata?

Metadata is **data about data**. It describes database objects such as tables, columns, data types, constraints, indexes, and permissions.

Q7. What is program-data independence?

It is the ability to make certain changes to the database's storage or structure without requiring corresponding changes to application programs.

Q8. What is a database view?

A view is a virtual representation of data defined by a SQL query. It can provide users with a customized or restricted view of the underlying data.

Q9. Why is concurrency control important?

> Concurrency control helps multiple users or transactions access shared data safely without producing incorrect or conflicting results.

Practice & Hands-On Exercises

Beginner Practice

1. Explain data redundancy with a simple example.
2. What is data integrity?
3. What is data security?
4. What is the difference between backup and recovery?
5. Why is data privacy important?

Think and Answer

1. Why is unnecessary duplicate data a problem?
2. Give an example of inconsistent data in an online shopping system.
3. Why should a student not have the same database permissions as an administrator?
4. How does a database support multiple users at the same time?
5. What is metadata? Give two examples.

SQL Connection

Consider:

```sql
CREATE VIEW Student_Public_Info AS
SELECT Student_ID, Name, Department
FROM Students;
```

1. What is the purpose of this view?
2. Which columns are visible through the view?
3. Why might a university use a view instead of giving every user access to the entire `Students` table?

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
GOOD DATABASE
     ↓
LESS REDUNDANCY
+ DATA INTEGRITY
+ DATA CONSISTENCY
+ SECURITY
+ PRIVACY
+ BACKUP & RECOVERY
+ CONTROLLED SHARING
     ↓
RELIABLE DATA MANAGEMENT
```

> ***A good database system does more than store data. It keeps data accurate, secure, consistent, recoverable, and available to authorized users.***

End of Core Database Characteristics

3.1 Database Transactions

Level: 3 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **database transaction** is a group of one or more database operations that are treated as **one logical unit of work**.

For example, transferring ₹1,000 from one bank account to another requires two operations:

1. Subtract ₹1,000 from Account 1.
2. Add ₹1,000 to Account 2.

These two operations should be treated as **one transaction**.

```text
TRANSFER ₹1,000
       ↓
Subtract from Account 1
       +
Add to Account 2
       ↓
ONE TRANSACTION
```

Copy

The important idea is:

> *A transaction should leave the database in a correct state.*

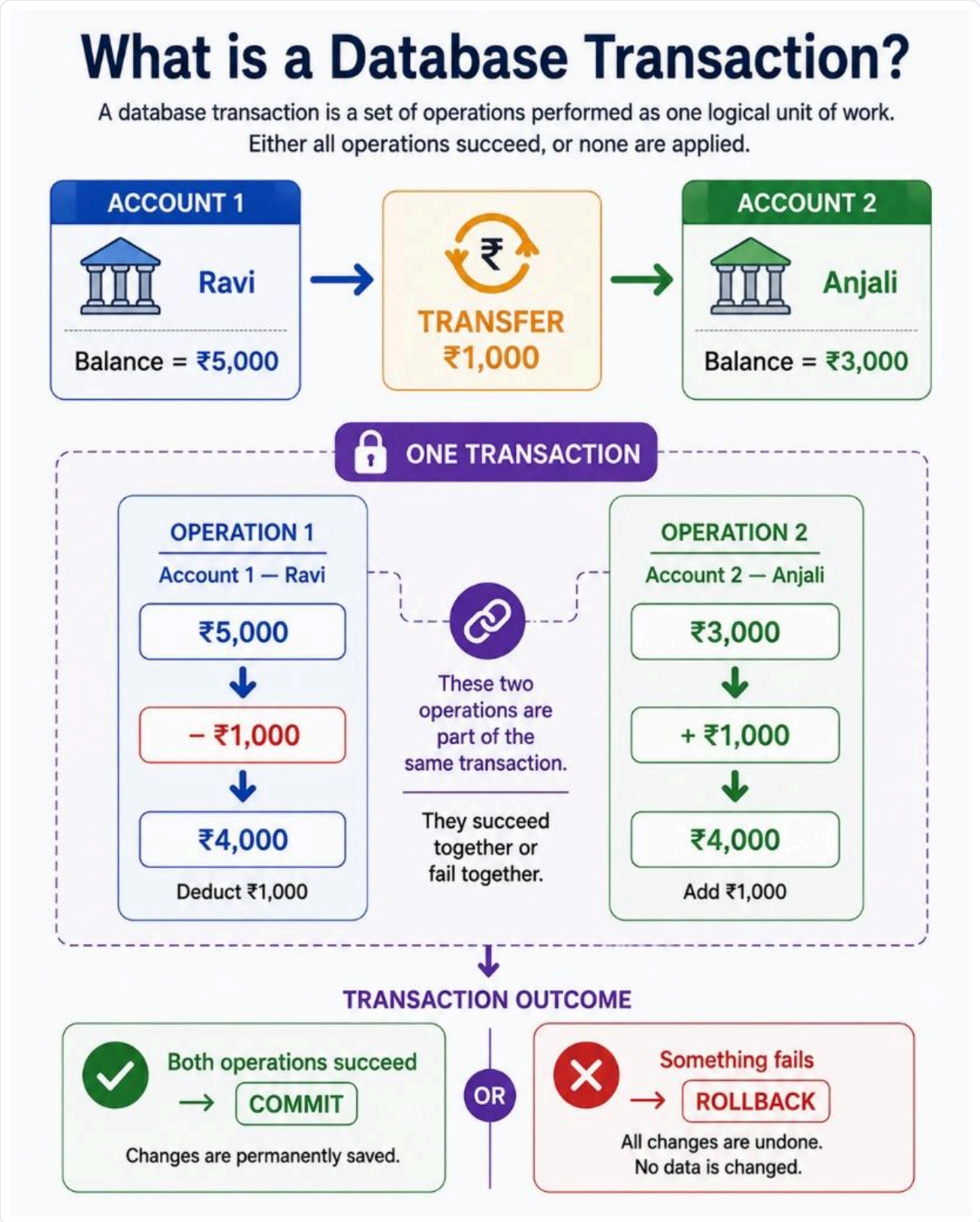


Figure 1: A database transaction groups related operations into one logical unit of work.

Why Do We Need Transactions?

Suppose Account 1 has ₹5,000 and Account 2 has ₹3,000.

We want to transfer ₹1,000.

Before the transfer:

```
TEXT

Ravi   → ₹5,000
Anjali → ₹3,000
```

Copy

After a successful transfer:

```
TEXT

Ravi   → ₹4,000
Anjali → ₹4,000
```

Copy

Now imagine the first operation succeeds but the second operation fails.

The database could incorrectly end up with:

```
TEXT

Ravi   → ₹4,000
Anjali → ₹3,000
```

Copy

₹1,000 has disappeared.

A transaction helps prevent this kind of **partial update**.

> ***Either the complete transaction succeeds, or the database should undo the changes when the transaction cannot be completed.***

Transaction Commands

Common SQL transaction commands include:

BEGIN

Starts a transaction.

```sql
BEGIN;
```

COMMIT

Permanently saves the changes made by the transaction.

```sql
COMMIT;
```

ROLLBACK

Undoes changes made during the current transaction that have not been committed.

```sql
ROLLBACK;
```

The basic flow is:

```text
BEGIN
  ↓
SQL OPERATIONS
  ↓
SUCCESS?
 ↙     ↘
YES     NO
 ↓       ↓
COMMIT  ROLLBACK
```

Practice Table Setup

We will use a simple `accounts` table.

```sql
CREATE TABLE accounts (
    account_id INT PRIMARY KEY,
    account_holder VARCHAR(50),
    balance NUMERIC(10, 2)
);

INSERT INTO accounts (account_id, account_holder, balance) VALUES
(1, 'Ravi Kumar', 5000.00),
(2, 'Anjali Mehta', 3000.00);
```

The table initially contains:

account_id	account_holder	balance
1	Ravi Kumar	5000.00
2	Anjali Mehta	3000.00

Example: Successful Transaction

To transfer ₹1,000 from Ravi to Anjali:

```sql
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 1000
WHERE account_id = 2;

COMMIT;
```

After the transaction:

account_id	account_holder	balance
1	Ravi Kumar	4000.00
2	Anjali Mehta	4000.00

Both updates were successfully completed and then committed.

Example: Rolling Back a Transaction

Suppose we start a transfer but decide not to save it.

```sql
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 1000
WHERE account_id = 2;

ROLLBACK;
```

The changes made during the transaction are undone.

The balances return to:

account_id	account_holder	balance
1	Ravi Kumar	5000.00
2	Anjali Mehta	3000.00

Transaction States

A transaction typically moves through a small number of states.

```text
START
  ↓
ACTIVE
  ↓
COMMIT → COMPLETED
  or
ROLLBACK → CANCELLED
```

For beginners, the important idea is:

- **Active** → transaction is being executed.

- **Committed** → changes are successfully saved.
- **Rolled back** → changes are undone.

What Happens When Something Fails?

Consider:

```sql
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE account_id = 1;

-- Second operation fails

ROLLBACK;
```

The first update should not remain as a partial transfer.

The rollback returns the database to the state before the transaction started.

This is one of the reasons transactions are essential in systems such as:

- Banking
- Payments
- Ticket booking
- E-commerce
- Inventory management

Transactions and ACID

Transactions are closely connected with the **ACID properties**:

```
TEXT

A → Atomicity
C → Consistency
I → Isolation
D → Durability
```

You will learn these properties in detail in the next topic.

For now, remember:

> ***ACID properties help transactions execute reliably and keep the database in a trustworthy state.***

Real-World Example: Online Ticket Booking

Imagine two people trying to book the same last available seat.

A transaction can group the required operations:

```
TEXT

Check Seat
   ↓
Reserve Seat
   ↓
Create Booking
   ↓
Record Payment
   ↓
COMMIT
```

If an important step fails:

```
TEXT

FAILURE
   ↓
ROLLBACK
   ↓
Undo incomplete changes
```

Copy

This prevents the system from creating an incomplete booking.

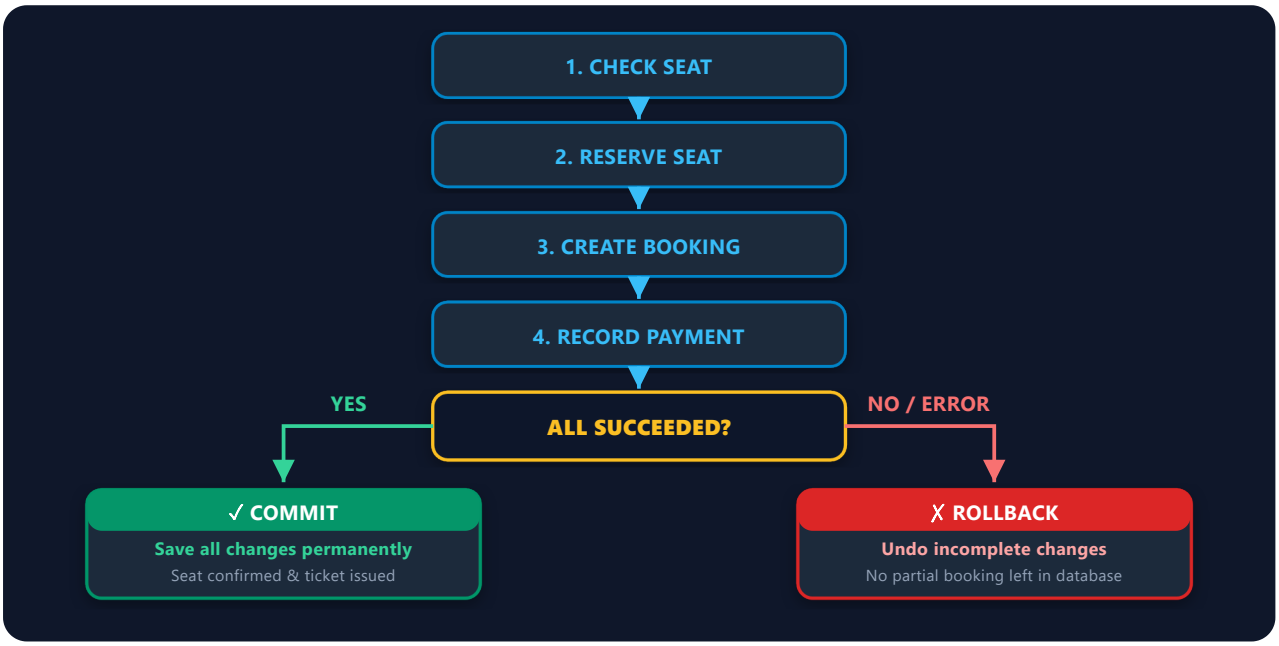


Figure 2: A transaction can group several related operations and commit them together or roll them back when the transaction fails.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
A transaction is always one SQL statement	A transaction can contain one or many SQL statements
`BEGIN` permanently saves changes	`BEGIN` starts the transaction; `COMMIT` saves it
`ROLLBACK` deletes all database data	It undoes uncommitted changes made by the current transaction
Every SQL query automatically needs manual `BEGIN` and `COMMIT`	Transaction behavior depends on the database and client/application settings
A transaction is only useful for banking	Transactions are useful anywhere multiple related operations must remain consistent
`COMMIT` can always be undone with `ROLLBACK`	Once changes are committed, a normal rollback cannot undo them

Placement Quick Points

```text
TRANSACTION
→ GROUPS RELATED OPERATIONS

BEGIN
→ START TRANSACTION

COMMIT
→ SAVE CHANGES

ROLLBACK
→ UNDO UNCOMMITTED CHANGES

TRANSACTION
→ PREVENTS PARTIAL WORK FROM BEING ACCEPTED
```

Copy

- A transaction is a logical unit of work.
- A transaction can contain one or more SQL operations.
- `BEGIN` starts a transaction.
- `COMMIT` saves the transaction's changes.
- `ROLLBACK` undoes uncommitted changes.
- Transactions are important when several operations must succeed together.
- ACID properties provide the foundation for reliable transaction processing.

Interview Questions

Q1. What is a database transaction?

> A transaction is a group of one or more database operations treated as a single logical unit of work.

Q2. Why are transactions important?

Transactions help ensure that related operations are completed reliably without leaving incorrect partial changes in the database.

Q3. What is the difference between COMMIT and ROLLBACK?

`COMMIT` permanently saves the changes made by the transaction, while `ROLLBACK` undoes uncommitted changes.

Q4. What does BEGIN do?

`BEGIN` starts a transaction so subsequent operations can be committed or rolled back as one unit.

Q5. Give a real-world example of a transaction.

A bank transfer is a common example. Money is deducted from one account and added to another. Both operations should succeed together.

Q6. What happens if one operation in a transaction fails?

The application can roll back the transaction so that incomplete changes are not left in the database.

Q7. Can a transaction contain multiple SQL statements?

Yes. A transaction can contain multiple operations such as `INSERT`, `UPDATE`, and `DELETE`.

Q8. What is ACID?

ACID stands for:

`text Atomicity Consistency Isolation Durability`

These properties help ensure reliable transaction processing.

Practice & Hands-On Exercises

Beginner Practice

1. Define a database transaction in your own words.
2. Why is a bank transfer a good example of a transaction?
3. What is the purpose of `BEGIN`?
4. What is the purpose of `COMMIT`?
5. What is the purpose of `ROLLBACK`?

Hands-On

Using the `accounts` table:

1. Start a transaction and subtract ₹500 from Ravi's account:

```text
BEGIN;

UPDATE accounts
SET balance = balance - 500
WHERE account_id = 1;
```

1. Add ₹500 to Anjali's account:

```sql
UPDATE accounts
SET balance = balance + 500
WHERE account_id = 2;
```

1. Commit the transaction:

```sql
COMMIT;
```

1. Check both balances:

```sql
SELECT * FROM accounts;
```

1. Repeat the transfer but use `ROLLBACK` instead of `COMMIT`. What happens to the balances?

Think and Answer

1. What could go wrong if a money transfer were treated as two independent operations?
2. Why should booking a last available seat and creating the booking record be handled carefully as related operations?
3. Can a transaction contain only one SQL statement? Explain.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```sql
BEGIN
   ↓
RELATED SQL OPERATIONS
   ↓
SUCCESS → COMMIT
   ↓
SAVE CHANGES

FAILURE
   ↓
ROLLBACK
   ↓
UNDO UNCOMMITTED CHANGES
```

> ***A** database transaction groups related operations into one logical unit so the database can commit successful work or roll back incomplete work.*

End of Database Transactions

3.2 ACID Properties

Level: 3 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

ACID properties help ensure that database transactions are **reliable, correct, and safe**.

ACID stands for:

```text
A → Atomicity
C → Consistency
I → Isolation
D → Durability
```

A simple way to remember them:

> ***A**tomicity* → *All or nothing*
> ***Consistency*** → *Valid data*
> ***Isolation*** → *Transactions work independently*
> ***Durability*** → *Committed changes stay saved*

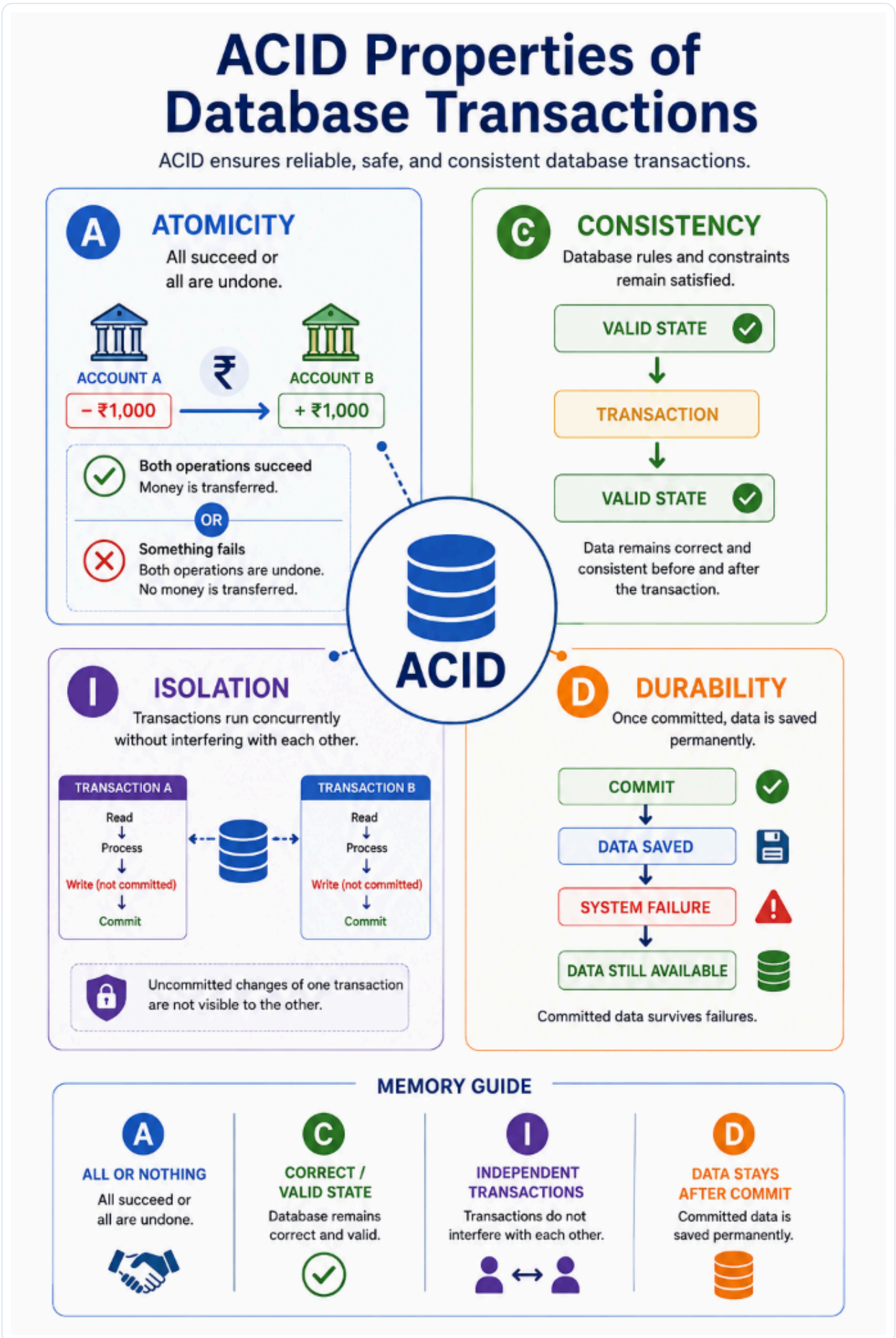

ACID Properties of Database Transactions
ACID ensures reliable, safe, and consistent database transactions.
A
ATOMICITY
All succeed or all are undone.
ACCOUNT A
₹
ACCOUNT B
– ₹1,000
+ ₹1,000
Both operations succeed
Money is transferred.
OR
Something fails
Both operations are undone.
No money is transferred.
C
CONSISTENCY
Database rules and constraints remain satisfied.
VALID STATE
TRANSACTION
VALID STATE
Data remains correct and consistent before and after the transaction.
ACID
I
ISOLATION
Transactions run concurrently without interfering with each other.
TRANSACTION A
Read
Process
Write (not committed)
Commit
TRANSACTION B
Read
Process
Write (not committed)
Commit
Uncommitted changes of one transaction are not visible to the other.
D
DURABILITY
Once committed, data is saved permanently.
COMMIT
DATA SAVED
SYSTEM FAILURE
DATA STILL AVAILABLE
Committed data survives failures.
MEMORY GUIDE
A
ALL OR NOTHING
All succeed or all are undone.
C
CORRECT / VALID STATE
Database remains correct and valid.
I
INDEPENDENT TRANSACTIONS
Transactions do not interfere with each other.
D
DATA STAYS AFTER COMMIT
Committed data is saved permanently.

Figure 1: ACID properties help database transactions remain reliable and preserve correct data.

Atomicity

Atomicity means a transaction is treated as **one complete unit of work**.

Either all required operations succeed, or the transaction is rolled back.

Consider a bank transfer:

```text
Subtract ₹1,000 from Ravi
        +
Add ₹1,000 to Anjali
        ↓
   ONE TRANSACTION
```

If the second operation fails, the first change should also be undone.

Example

```sql
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 1000
WHERE account_id = 2;

ROLLBACK;
```

After `ROLLBACK`, both changes are undone.

> *Atomicity = All or nothing.*

Consistency

Consistency means a transaction should move the database from **one valid state to another valid state** while preserving defined rules and constraints.

For example, suppose an account cannot have an invalid negative balance according to the application's rules.

A successful transaction should not leave the database in an invalid state.

```text
VALID STATE
    ↓
TRANSACTION
    ↓
VALID STATE
```

Consistency is maintained through database constraints, transaction rules, application logic, and other controls.

> *Consistency = The database remains valid before and after the transaction.*

Isolation

Isolation means concurrent transactions should not improperly interfere with each other.

Imagine two transactions running at the same time:

```
TEXT

Transaction A
    ↓
Database

Transaction B
    ↓
Database
```

Copy

The database system manages their interaction so that intermediate work from one transaction does not incorrectly affect another.

For example, a transaction may make changes internally before it commits. Other transactions should see data according to the database's isolation rules, rather than seeing incomplete work.

> ***Isolation = Concurrent transactions are controlled so they do not produce incorrect interference.***

Durability

Durability means that once a transaction is successfully **committed**, its changes are preserved even if a system failure occurs afterward.

For example:

```text
TRANSACTION
    ↓
COMMIT
    ↓
DATA SAVED
    ↓
SYSTEM RESTART
    ↓
COMMITTED DATA REMAINS
```

Database systems use mechanisms such as persistent storage and recovery techniques to provide durability.

> ***Durability** = Once committed, changes are not lost because of a normal system restart or crash.*

ACID in One Example

Consider transferring ₹1,000 from Ravi to Anjali.

```text
BEGIN
  ↓
Subtract ₹1,000 from Ravi
  ↓
Add ₹1,000 to Anjali
  ↓
COMMIT
```

The four properties apply as follows:

Property	What It Means in This Transfer
Atomicity	Both account updates succeed together or are rolled back
Consistency	Account and database rules remain valid
Isolation	Other transactions do not improperly see incomplete transfer work
Durability	After commit, the transfer remains saved even after a failure

Practice Table

This topic uses the `accounts` table from **3.1 Database Transactions**.

If the table is not already available, use:

```sql
CREATE TABLE accounts (
    account_id INT PRIMARY KEY,
    account_holder VARCHAR(50),
    balance NUMERIC(10, 2)
);

INSERT INTO accounts (account_id, account_holder, balance) VALUES
(1, 'Ravi Kumar', 5000.00),
(2, 'Anjali Mehta', 3000.00);
```

Example: Atomicity in Action

Start a transaction and perform both account updates:

```sql
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 1000
WHERE account_id = 2;

ROLLBACK;
```

Because the transaction is rolled back, **both updates are undone**.

The balances remain:

account_id	account_holder	balance
1	Ravi Kumar	5000.00
2	Anjali Mehta	3000.00

This demonstrates the **all-or-nothing** nature of atomicity.

COMMIT vs ROLLBACK

The difference is important:

```text
COMMIT
→ Make the transaction's changes permanent

ROLLBACK
→ Undo uncommitted changes
```

Example:

```sql
BEGIN;

UPDATE accounts
SET balance = balance - 500
WHERE account_id = 1;

COMMIT;
```

The change is committed and becomes part of the database state.

With:

```sql
ROLLBACK;
```

uncommitted changes made during the transaction are undone.

Why ACID Matters

ACID properties are especially important when incorrect or partial updates could cause serious problems.

Examples include:

- Banking
- Payment processing
- Ticket booking
- Inventory management
- E-commerce orders

For example, a payment system should not record a successful payment while failing to correctly update the related order state.

ACID is important wherever database operations must be reliable and consistent.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Atomicity means only one SQL statement can run	A transaction can contain multiple statements
Consistency means every user always sees the latest data	Consistency is about maintaining valid database states and rules
Isolation means transactions never run at the same time	Transactions can run concurrently; the DBMS controls their interaction
Durability means data can never be deleted	Durability means committed changes survive failures according to the database's durability guarantees
`ROLLBACK` undoes committed changes	Normal rollback only undoes uncommitted work in the current transaction
ACID is only useful for banking	ACID is useful in many systems where reliable transactions matter

Placement Quick Points

```text
A → ATOMICITY
  → ALL OR NOTHING

C → CONSISTENCY
  → VALID STATE BEFORE AND AFTER

I → ISOLATION
  → CONTROLLED CONCURRENT TRANSACTIONS

D → DURABILITY
  → COMMITTED DATA PERSISTS
```

Copy

- **Atomicity** ensures a transaction is treated as one unit of work.
- **Consistency** keeps the database within its defined valid rules and constraints.
- **Isolation** controls how concurrent transactions interact.
- **Durability** preserves committed changes after system failures.
- ACID properties are fundamental to reliable transaction processing.
- `COMMIT` saves a transaction's changes.
- `ROLLBACK` undoes uncommitted changes.

Interview Questions

Q1. What does ACID stand for?

ACID stands for:

`text Atomicity Consistency Isolation Durability`

These properties help make database transactions reliable.

Q2. What is Atomicity?

Atomicity means that all required operations in a transaction succeed together, or the transaction is rolled back.

Q3. What is Consistency?

Consistency means a successful transaction leaves the database in a valid state that satisfies its defined rules and constraints.

Q4. What is Isolation?

Isolation means concurrent transactions are controlled so that one transaction does not improperly interfere with another.

Q5. What is Durability?

Durability means that once a transaction is committed, its changes remain stored even if a system failure occurs afterward.

Q6. Give a real-world example of Atomicity.

A bank transfer is a common example. The money must be deducted from one account and added to another as one logical operation.

Q7. Why is Isolation important?

Without proper isolation, concurrent transactions could see or depend on incomplete changes and produce incorrect results.

Q8. What happens to a transaction after COMMIT?

Its changes are committed to the database and become part of the durable database state.

Q9. What is the difference between COMMIT and ROLLBACK?

`COMMIT` saves the transaction's changes. `ROLLBACK` undoes uncommitted changes made by the transaction.

Practice & Hands-On Exercises

Beginner Practice

1. What does each letter in ACID stand for?
2. Explain Atomicity using a bank-transfer example.
3. Explain Consistency in your own words.
4. Why is Isolation important when many users access the same database?
5. What does Durability guarantee after a successful `COMMIT`?

Hands-On

Using the `accounts` table:

1. Start a transaction and transfer ₹500 from Ravi to Anjali:

```text
BEGIN;

UPDATE accounts
SET balance = balance - 500
WHERE account_id = 1;

UPDATE accounts
SET balance = balance + 500
WHERE account_id = 2;

COMMIT;
```

1. Verify both balances:

```sql
SELECT * FROM accounts;
```

1. Start another transaction and transfer ₹1,000.
2. Roll back the transaction and check whether the balances changed.

3. Explain which ACID property is demonstrated by the rollback.

Think and Answer

1. What could happen if Atomicity were not maintained during a bank transfer?
2. Why should a committed payment remain available after a server restart?
3. Give one example where two transactions may run concurrently.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```SQL
A → ALL OR NOTHING
C → VALID DATABASE STATE
I → CONTROLLED CONCURRENCY
D → COMMITTED DATA PERSISTS
```

ACID properties make database transactions reliable by ensuring complete execution, valid database states, controlled concurrent access, and persistence of committed changes.

End of ACID Properties

3.3 Normalization

Level: 3 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Normalization is the process of organizing data in a relational database to **reduce unnecessary duplication and avoid data-related problems**.

It generally involves dividing a large table into smaller, related tables and connecting them using keys.

Normalization helps reduce:

- **Data redundancy** — unnecessary duplicate data
- **Update anomalies** — the same fact must be updated in multiple places
- **Insert anomalies** — a new fact cannot be added without unrelated data
- **Delete anomalies** — deleting one record accidentally removes another useful fact

A simple idea is:

```text
ONE LARGE TABLE
      ↓
REMOVE REPETITION
      ↓
SEPARATE RELATED DATA
      ↓
CONNECTED TABLES
```

> *Normalization organizes relational data so that each fact is stored in an appropriate place and unnecessary duplication is reduced.*

Why Is Normalization Needed?

Imagine an employee table like this:

Emp_ID	Emp_Name	Department	Dept_Location	Emp_Skills
101	Nick Wise	HR	London	Recruitment, Payroll
102	John Cader	Finance	Australia	Budgeting
103	Lily Case	HR	London	Recruitment
104	Ford Dawid	IT	Chicago	Programming, Testing

At first, this looks simple.

But notice:

- `HR` and `London` are repeated.
- An employee can have multiple skills in one cell.
- Department information is mixed with employee information.
- Updating department details can require changing multiple rows.

These are signs that the table can be better organized.

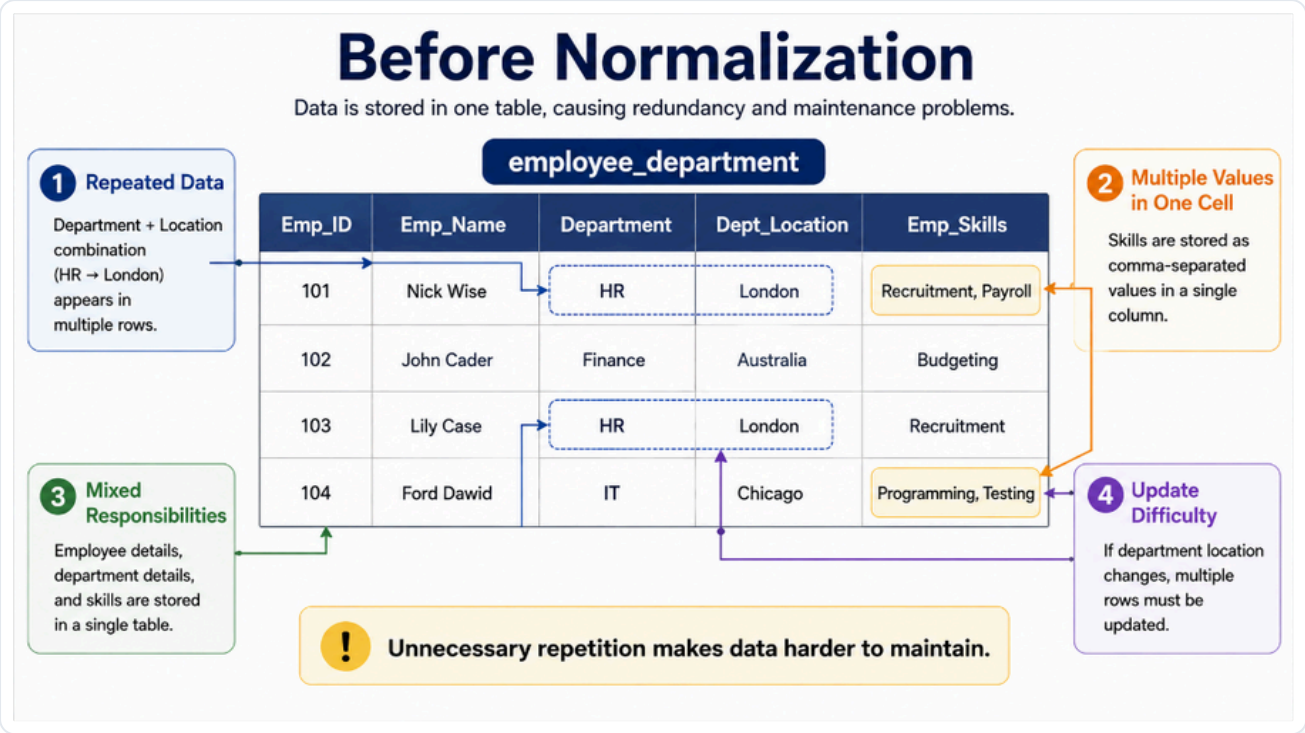


Figure 1: The unnormalized table contains repeated information and multiple values in a single cell, making the data harder to maintain.

Problems in Unnormalized Data

1. Data Redundancy

The same information is stored more than once.

For example:

```text
HR → London
HR → London
```

The department location is repeated for every HR employee.

2. Update Anomaly

Suppose the HR department moves from London to Manchester.

We may need to update multiple employee rows.

If one row is missed:

```text
Employee 101 → HR → Manchester
Employee 103 → HR → London
```

The database now contains conflicting information.

> *Update anomaly occurs when the same fact must be changed in multiple places and some copies are missed.*

---

3. Insert Anomaly

Suppose a new department is created, but it does not yet have any employees.

In the original table, department information depends on having an employee row.

We may be forced to insert employee-related data just to store the department.

> *Insert anomaly occurs when one fact cannot be inserted independently because another unrelated fact is required.*

---

4. Delete Anomaly

Suppose the last employee in the Finance department leaves.

If we delete that employee's row, we may also lose the only stored information about the Finance department.

> ***Delete anomaly occurs when deleting one fact unintentionally removes another useful fact.***

Practice Table Setup

The original table can be created as:

```SQL
CREATE TABLE employee_department (
    emp_id INT,
    emp_name VARCHAR(50),
    department VARCHAR(50),
    dept_location VARCHAR(50),
    emp_skills VARCHAR(100)
);

INSERT INTO employee_department
(emp_id, emp_name, department, dept_location, emp_skills)
VALUES
(101, 'Nick Wise', 'HR', 'London', 'Recruitment, Payroll'),
(102, 'John Cader', 'Finance', 'Australia', 'Budgeting'),
(103, 'Lily Case', 'HR', 'London', 'Recruitment'),
(104, 'Ford Dawid', 'IT', 'Chicago', 'Programming, Testing');
```

The main problem is:

```
TEXT

emp_skills
→ "Recruitment, Payroll"
```

Copy

Multiple skills are stored inside one cell.

This is not a clean relational design.

First Normal Form (1NF)

The first important step in normalization is **First Normal Form (1NF)**.

A table is in 1NF when each column contains **atomic values**—one value in each cell—and repeating groups are removed.

For example, instead of:

```
TEXT

Emp_ID = 101
Skills = Recruitment, Payroll
```

Copy

we store the skills separately:

Emp_ID	Skill
101	Recruitment
101	Payroll

Now each cell contains one value.

> *1NF = One value per cell, with no repeating groups.*

Normalized Design

We can separate the original table into related tables.

Department

Dept_ID	Department	Dept_Location
D1	HR	London
D2	Finance	Australia
D3	IT	Chicago

Employee

Emp_ID	Emp_Name	Dept_ID
101	Nick Wise	D1
102	John Cader	D2
103	Lily Case	D1
104	Ford Dawid	D3

Employee_Skills

Emp_ID	Emp_Skill
101	Recruitment
101	Payroll
102	Budgeting
103	Recruitment
104	Programming
104	Testing

Now:

- Department information is stored once.
- Employee information is stored separately.
- Skills are stored as individual values.
- The tables are connected using keys.

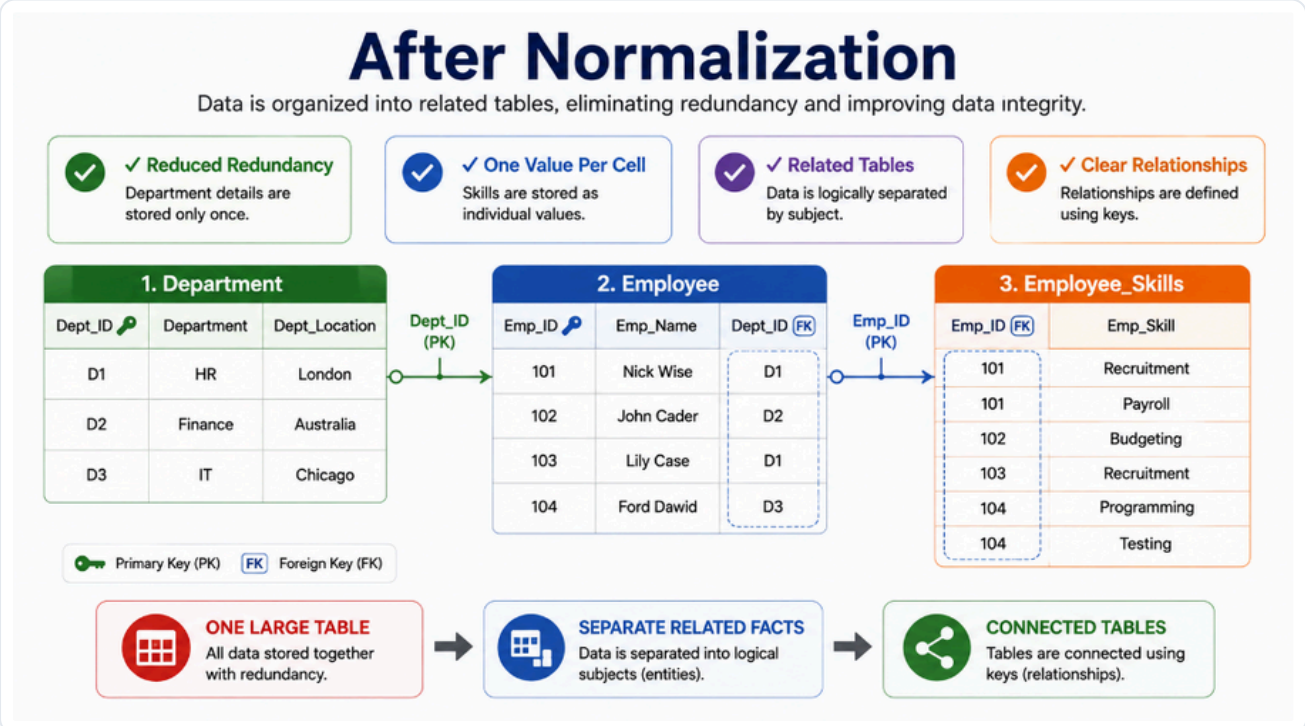


Figure 2: Normalization separates different facts into related tables, reducing unnecessary repetition and improving data organization.

Normalization Transformation

The transformation can be summarized as:

```text
UNNORMALIZED TABLE
        ↓
Remove repeating values
        ↓
Separate related facts
        ↓
Create relationships
        ↓
NORMALIZED TABLES
```

Before normalization:

```text
Employee + Department + Skills
            ↓
        ONE TABLE
```

After normalization:

```text
Department
     ↓
Employee
     ↓
Employee_Skills
```

The goal is **not simply to create more tables**.

The goal is to store each fact in the appropriate place and establish meaningful relationships between the tables.

2NF and 3NF — Basic Idea

Normalization is commonly discussed through several normal forms.

First Normal Form — 1NF

Focuses on **atomic values** and removing repeating groups.

```
TEXT
Copy

1NF → ONE VALUE PER CELL
```

Second Normal Form — 2NF

Builds on 1NF and ensures that non-key attributes depend on the **whole primary key**, particularly when a table has a composite key.

```
TEXT
Copy

2NF → NO PARTIAL DEPENDENCY
```

Third Normal Form — 3NF

Builds on 2NF and removes **transitive dependencies**, so non-key attributes depend on the key rather than on another non-key attribute.

```
TEXT
Copy

3NF → NO TRANSITIVE DEPENDENCY
```

For most beginner database designs, reaching an appropriate **3NF** structure is a common normalization goal.

> *1NF → Atomic values*
> *2NF → Full dependency on the key*
> *3NF → No transitive dependency*

SQL Example

The normalized tables can be created using:

```sql
SQL
Copy

CREATE TABLE department (
    dept_id VARCHAR(5) PRIMARY KEY,
    department VARCHAR(50),
    dept_location VARCHAR(50)
);

CREATE TABLE employee (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    dept_id VARCHAR(5),
    FOREIGN KEY (dept_id) REFERENCES department(dept_id)
);

CREATE TABLE employee_skills (
    emp_id INT,
    emp_skill VARCHAR(50),
    FOREIGN KEY (emp_id) REFERENCES employee(emp_id)
);
```

Now the relationships are explicitly represented using **foreign keys**.

For example:

```
employee.dept_id
        ↓
department.dept_id
```

Benefits of Normalization

Normalization can help:

- Reduce unnecessary duplication
- Prevent update anomalies
- Prevent insert anomalies
- Prevent delete anomalies
- Improve data integrity
- Make relationships clearer
- Make database maintenance easier

However, normalization can also create **more tables and more joins**. In some systems, deliberate denormalization may be used for performance or reporting requirements.

> *Good database design balances data integrity, maintainability, and performance.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Normalization means creating as many tables as possible	Tables should be separated based on meaningful data dependencies
Normalization completely removes all duplicate values	It reduces unnecessary redundancy; some repetition may still be intentional
1NF means no duplicate rows	1NF mainly requires atomic values and removal of repeating groups
Every database must always be fully normalized	Design depends on application requirements; some systems intentionally denormalize
More tables automatically means better design	Excessive decomposition can make queries more complex
Normalization is only about performance	Its main goals are data organization, integrity, and reducing anomalies
A foreign key stores the related table	A foreign key stores a value that references a key in another table

Placement Quick Points

```text
NORMALIZATION
→ ORGANIZE RELATIONAL DATA

1NF
→ ATOMIC VALUES
→ ONE VALUE PER CELL

2NF
→ NO PARTIAL DEPENDENCY

3NF
→ NO TRANSITIVE DEPENDENCY

MAIN GOALS
→ REDUCE REDUNDANCY
→ AVOID INSERT / UPDATE / DELETE ANOMALIES
→ IMPROVE DATA INTEGRITY
```

- Normalization organizes relational data into appropriate related tables.
- It reduces unnecessary data redundancy.
- It helps prevent insertion, update, and deletion anomalies.
- **1NF** focuses on atomic values and repeating groups.
- **2NF** removes partial dependency on part of a composite key.
- **3NF** removes transitive dependency.
- Normalization improves maintainability and integrity, but excessive normalization can increase query complexity.

Interview Questions

Q1. What is normalization?

Normalization is the process of organizing relational data to reduce unnecessary redundancy and prevent data anomalies.

Q2. Why is normalization needed?

It helps reduce duplicate data and prevents insertion, update, and deletion anomalies.

Q3. What is data redundancy?

Data redundancy is the unnecessary repetition of the same data in multiple places.

Q4. What is an update anomaly?

An update anomaly occurs when the same fact is stored in multiple locations and updating only some of those locations creates inconsistent data.

Q5. What is an insert anomaly?

An insert anomaly occurs when a new fact cannot be added without also providing unrelated information.

Q6. What is a delete anomaly?

A delete anomaly occurs when deleting one record unintentionally removes another useful fact.

Q7. What is 1NF?

1NF requires atomic values and removes repeating groups, so each cell contains a single value.

Q8. What is 2NF?

2NF builds on 1NF and requires non-key attributes to depend on the whole primary key, eliminating partial dependency.

Q9. What is 3NF?

3NF builds on 2NF and removes transitive dependencies so non-key attributes depend on the key rather than on other non-key attributes.

Q10. Does normalization always improve query performance?

> No. Normalization mainly improves data organization and integrity. It can sometimes require more joins, so performance depends on the workload and design.

Practice & Hands-On Exercises

Beginner Practice

1. Define normalization in your own words.
2. What is data redundancy?
3. Explain update, insert, and delete anomalies.
4. What does 1NF mean?
5. What is the difference between 2NF and 3NF?

Table Analysis

Consider:

Emp_ID	Emp_Name	Department	Skills
101	Nick	HR	Recruitment, Payroll
102	John	Finance	Budgeting

1. Identify the repeating/multiple-value problem.
2. Why is storing two skills in one cell undesirable?
3. Which normalization form addresses the atomic-value problem?

SQL Connection

Using the normalized tables:

1. Write a query to display each employee with their department:

```sql
SELECT e.emp_name, d.department, d.dept_location
FROM employee e
JOIN department d ON e.dept_id = d.dept_id;
```

1. Write a query to display each employee's skills:

```sql
SELECT e.emp_name, s.emp_skill
FROM employee e
JOIN employee_skills s ON e.emp_id = s.emp_id;
```

1. Explain how `employee.dept_id` connects to `department.dept_id`.
2. Explain how `employee_skills.emp_id` connects to `employee.emp_id`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
UNNORMALIZED DATA
        ↓
REDUNDANCY + ANOMALIES
        ↓
NORMALIZATION
        ↓
RELATED TABLES
        ↓
LESS REDUNDANCY
+ BETTER INTEGRITY
+ EASIER MAINTENANCE
```

> *Normalization organizes relational data into related tables so that unnecessary duplication is reduced and data anomalies are avoided.*

End of Normalization

Meptrasoft AI TECHNOLOGIES

3.4 Normal Forms

Level: 3 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Normal forms are levels of database design used in normalization.

Each normal form introduces additional rules for organizing relational data and reducing different types of redundancy or dependency problems.

The commonly discussed normal forms are:

```text
1NF
 ↓
2NF
 ↓
3NF
 ↓
BCNF
 ↓
4NF
 ↓
5NF
```

For SQL and database beginners, **1NF, 2NF, and 3NF** are the most important starting points.

> ***Each higher normal form builds on the rules of the previous one.***

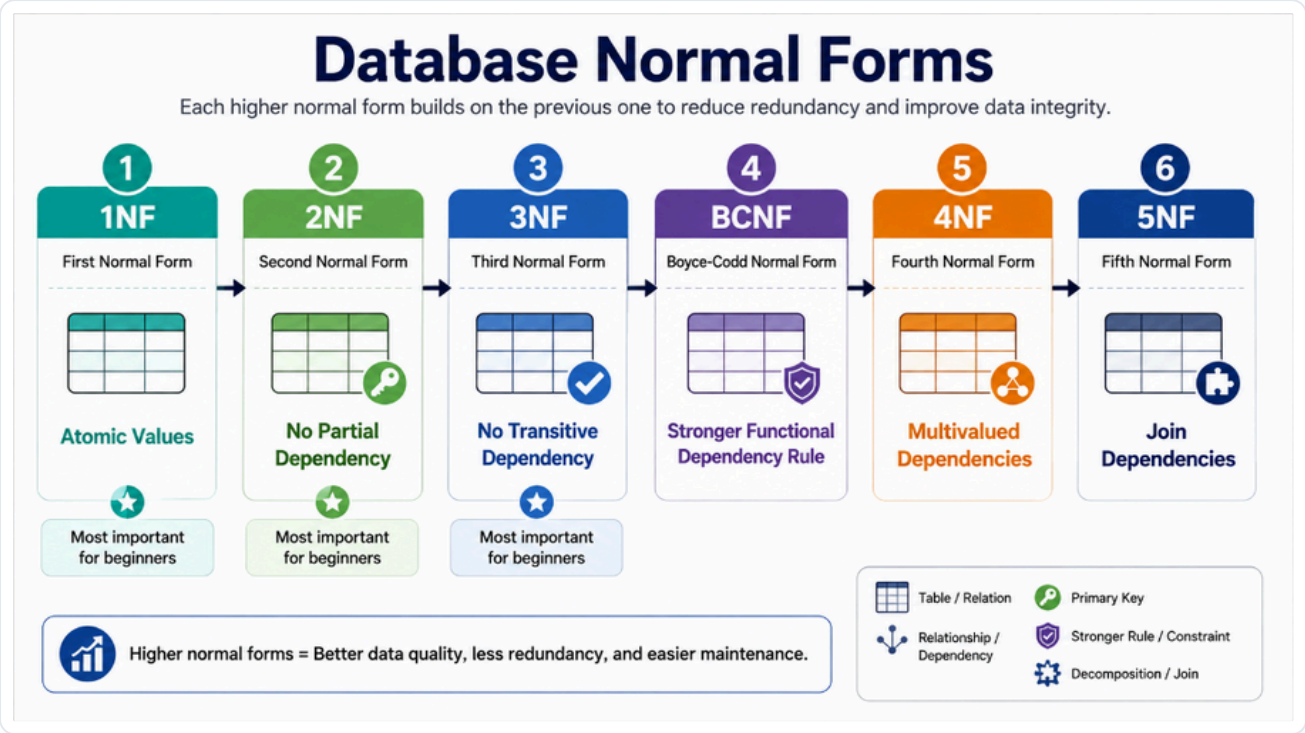


Figure 1: Normal forms provide progressively stronger rules for organizing relational database structures.

First Normal Form (1NF)

First Normal Form (1NF) requires each column value to be **atomic**.

In simple terms:

> *One cell should contain one value, not a list of values.*

For example, this violates 1NF:

Order_ID	Customer	Products
1	John Smith	Laptop, Mouse, Keyboard

The `Products` cell contains three values.

A better structure is:

Order_ID	Customer	Product
1	John Smith	Laptop
1	John Smith	Mouse
1	John Smith	Keyboard

Now each cell contains a single value.

What Does Atomic Mean?

Atomic means that a value is treated as one indivisible value for the purpose of that column.

For example:

```text
Good:
Product = Laptop

Problem:
Products = Laptop, Mouse, Keyboard
```

Similarly, this is problematic when a column stores multiple phone numbers in one field:

```text
Phone = 9876543210, 9123456780
```

A design that follows 1NF stores the values separately according to the data model.

> *1NF helps keep data in a form that can be stored, searched, and manipulated consistently.*

Violating 1NF — Practice Table Setup

Consider this table:

```sql
CREATE TABLE orders_unnormalized (
    order_id INT,
    customer_name VARCHAR(50),
    products VARCHAR(100),
    quantities VARCHAR(50)
);

INSERT INTO orders_unnormalized
(order_id, customer_name, products, quantities)
VALUES
(1, 'John Smith', 'Laptop, Mouse, Keyboard', '1, 2, 1');
```

The data looks like:

order_id	customer_name	products	quantities
1	John Smith	Laptop, Mouse, Keyboard	1, 2, 1

The problem is that both `products` and `quantities` contain multiple values.

This violates the atomic-value requirement of 1NF.

Satisfying 1NF

We can represent each product as a separate row:

```sql
CREATE TABLE orders_normalized (
    order_id INT,
    customer_name VARCHAR(50),
    product VARCHAR(50),
    quantity INT
);

INSERT INTO orders_normalized
(order_id, customer_name, product, quantity)
VALUES
(1, 'John Smith', 'Laptop', 1),
(1, 'John Smith', 'Mouse', 2),
(1, 'John Smith', 'Keyboard', 1);
```

Now the table contains:

order_id	customer_name	product	quantity
1	John Smith	Laptop	1
1	John Smith	Mouse	2
1	John Smith	Keyboard	1

Each cell contains one value.

```text
MULTIPLE VALUES IN ONE CELL
            ↓
        1NF DESIGN
            ↓
ONE VALUE PER CELL
```

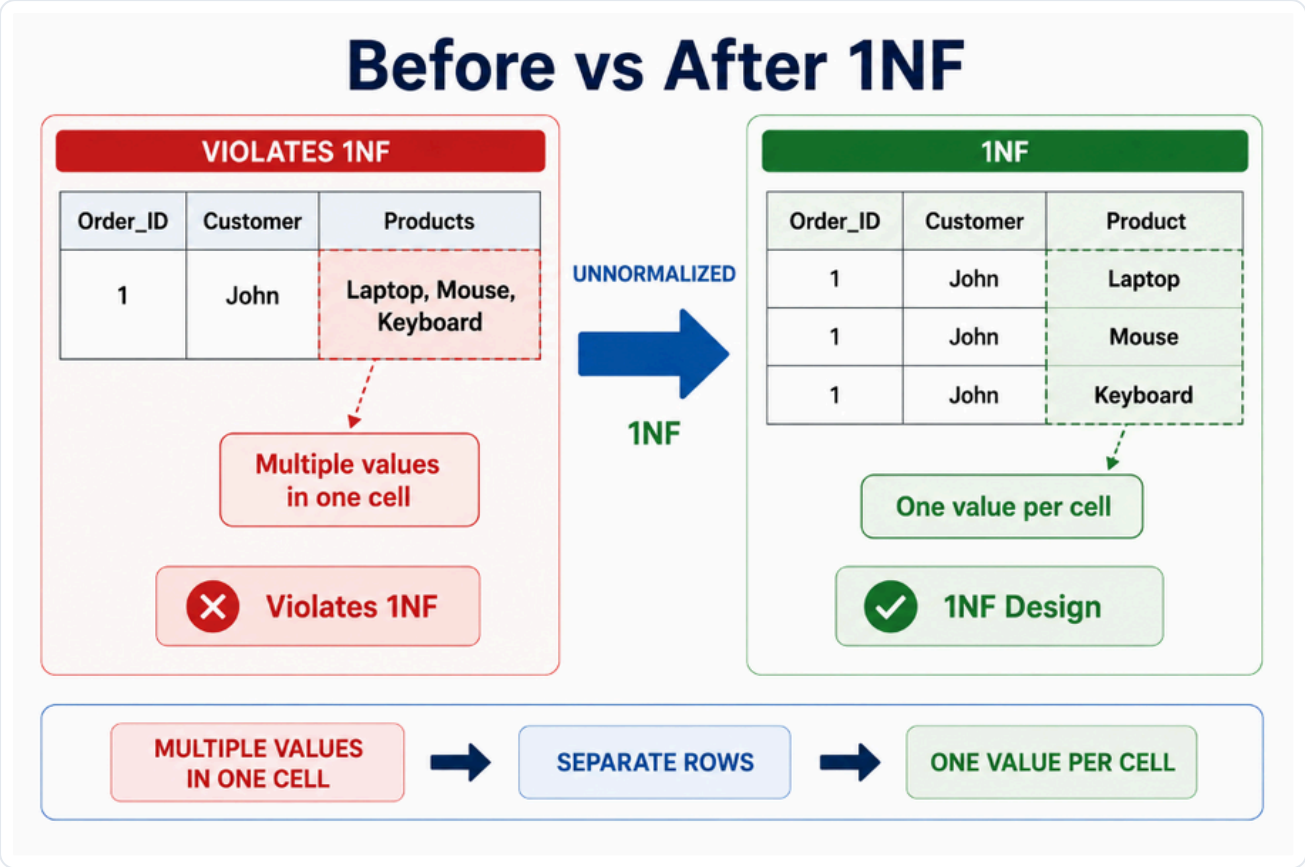


Figure 2: 1NF removes multiple values from a single cell and stores each value separately.

Second Normal Form (2NF)

Second Normal Form (2NF) builds on 1NF.

A table is in 2NF when:

1. It is already in **1NF**, and
2. Every non-key attribute depends on the **whole primary key**, not just part of a composite key.

The second rule is called removing **partial dependency**.

Simple Example

Consider:

Student_ID	Course_ID	Student_Name	Course_Name	Mark
101	C01	Arun	Python	85
101	C02	Arun	SQL	90

Suppose the primary key is:

```
TEXT

(Student_ID, Course_ID)
```

`Student_Name` depends only on `Student_ID`.

`Course_Name` depends only on `Course_ID`.

They do not depend on the **whole composite key**.

That is a partial dependency.

A better design separates the information:

```
TEXT

Students
Student_ID → Student_Name

Courses
Course_ID → Course_Name

Enrollments
Student_ID + Course_ID → Mark
```

> *2NF removes partial dependency on part of a composite key.*

Third Normal Form (3NF)

Third Normal Form (3NF) builds on 2NF.

A table is in 3NF when:

1. It is already in **2NF**, and
2. Non-key attributes do not depend on another non-key attribute.

This removes **transitive dependency**.

Simple Example

Consider:

Student_ID	Student_Name	Dept_ID	Department_Name
101	Arun	D1	Computer Science
102	Priya	D1	Computer Science

Here:

```text
Student_ID
   ↓
Dept_ID
   ↓
Department_Name
```

`Department_Name` depends on `Dept_ID`, rather than directly on the student key.

A normalized design can separate the department information:

```
TEXT

Students
Student_ID → Student_Name, Dept_ID

Departments
Dept_ID → Department_Name
```

Copy

> *3NF removes transitive dependency so non-key attributes depend on the key, not on another non-key attribute.*

BCNF

BCNF stands for **Boyce-Codd Normal Form**.

It is a stronger version of 3NF.

BCNF focuses on **functional dependencies** and requires that every determinant in a relation be a candidate key.

For beginner SQL learning, it is enough to remember:

> *BCNF applies stricter dependency rules than 3NF.*

Fourth Normal Form (4NF)

Fourth Normal Form (4NF) deals with **multivalued dependencies**.

It is useful when a table contains two or more independent multi-valued relationships that should not be stored together.

For example, if an employee independently has multiple skills and multiple languages, storing both lists together in one table can create unnecessary combinations.

4NF helps separate those independent multi-valued facts.

> ***4NF reduces unnecessary duplication caused by independent multi-valued relationships.***

Fifth Normal Form (5NF)

Fifth Normal Form (5NF) deals with **join dependencies**.

It focuses on situations where a table can be decomposed into smaller tables and reconstructed correctly through joins without introducing unwanted combinations or losing information.

For most beginner SQL work, 5NF is less commonly encountered than 1NF–3NF.

> ***5NF addresses complex join dependencies in relational designs.***

Normal Forms at a Glance

Normal Form	Main Focus
1NF	Atomic values
2NF	No partial dependency
3NF	No transitive dependency
BCNF	Stronger functional dependency rule
4NF	Multivalued dependencies
5NF	Join dependencies

Memory Trick

```text
1NF  → ONE VALUE
2NF  → WHOLE KEY
3NF  → ONLY THE KEY
BCNF → STRONGER KEY RULE
4NF  → MULTI-VALUED DEPENDENCIES
5NF  → JOIN DEPENDENCIES
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
1NF means there can be no duplicate values	1NF focuses on atomic values and repeating groups
2NF is simply "more normalized" than 1NF	2NF specifically addresses partial dependency and requires 1NF
3NF means there are no duplicate rows	3NF addresses transitive dependency
Every table automatically has a composite key	A composite key is a key made from multiple columns; many tables use a single-column key
BCNF is the same as 3NF	BCNF is a stricter normal form based on functional dependencies
Higher normal forms are always necessary	The required level depends on the database design and business requirements
More normalization always means better performance	Normalization can improve integrity but may introduce additional joins

Placement Quick Points

```text
1NF  → ATOMIC VALUES
2NF  → NO PARTIAL DEPENDENCY
3NF  → NO TRANSITIVE DEPENDENCY
BCNF → STRONGER FUNCTIONAL DEPENDENCY RULE
4NF  → NO UNNECESSARY MULTIVALUED DEPENDENCY
5NF  → JOIN DEPENDENCY
```

Copy

- **1NF** requires atomic values and removes repeating groups.
- **2NF** builds on 1NF and removes partial dependency on part of a composite key.
- **3NF** builds on 2NF and removes transitive dependency.
- **BCNF** applies a stronger rule to functional dependencies.
- **4NF** deals with multivalued dependencies.
- **5NF** deals with join dependencies.
- In practical SQL learning, **1NF, 2NF, and 3NF** are the most important normal forms to understand first.

Interview Questions

Q1. What is 1NF?

1NF requires each cell to contain an atomic value and removes repeating groups or multiple values stored together.

Q2. What is 2NF?

2NF requires a table to be in 1NF and ensures that non-key attributes depend on the entire primary key, eliminating partial dependency.

Q3. What is 3NF?

3NF requires a table to be in 2NF and removes transitive dependency so non-key attributes depend on the key rather than another non-key attribute.

Q4. What is BCNF?

BCNF is a stronger normal form than 3NF. It requires every determinant in a relation to be a candidate key.

Q5. What is the difference between 1NF, 2NF, and 3NF?

```text
1NF → Atomic values
2NF → No partial dependency
3NF → No transitive dependency
```

Each form builds on the previous one.

Q6. What is a partial dependency?

A partial dependency occurs when a non-key attribute depends on only part of a composite key instead of the complete key.

Q7. What is a transitive dependency?

A transitive dependency occurs when a non-key attribute depends on another non-key attribute rather than directly on the key.

Q8. Are 4NF and 5NF commonly required in everyday SQL development?

1NF, 2NF, and 3NF are much more commonly used as foundational normalization concepts. 4NF and 5NF are relevant for more specialized relational designs.

Q9. Does higher normalization always improve performance?

No. Normalization primarily improves organization and data integrity. Higher normalization can sometimes require more joins, so performance depends on the workload and database design.

Practice & Hands-On Exercises

Beginner Practice

1. What is the main purpose of 1NF?
2. What problem does 2NF solve?
3. What problem does 3NF solve?
4. What is BCNF?
5. What are 4NF and 5NF mainly concerned with?

Classify the Problem

Identify the normal form that addresses each situation:

1. `Products = 'Laptop, Mouse, Keyboard'` in one cell.
2. A non-key column depends only on part of a composite key.
3. A non-key column depends on another non-key column.
4. Two independent multi-valued relationships create unnecessary combinations.

SQL Connection

Consider:

Student_ID	Course_ID	Student_Name	Course_Name	Mark
101	C01	Arun	Python	85
101	C02	Arun	SQL	90

1. Identify the likely composite key.
2. Which columns depend only on `Student_ID`?
3. Which columns depend only on `Course_ID`?
4. How would you split this table to move toward 2NF?

CODE Copy

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50)
);

CREATE TABLE courses (
    course_id VARCHAR(5) PRIMARY KEY,
    course_name VARCHAR(50)
);

CREATE TABLE enrollments (
    student_id INT REFERENCES students(student_id),
    course_id VARCHAR(5) REFERENCES courses(course_id),
    mark INT,
    PRIMARY KEY (student_id, course_id)
);
```

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE Copy

```
1NF
→ ONE VALUE PER CELL
       ↓
2NF
→ FULL DEPENDENCY ON WHOLE KEY
       ↓
3NF
→ NO TRANSITIVE DEPENDENCY
       ↓
BCNF
→ STRONGER DEPENDENCY RULE
       ↓
4NF / 5NF
→ ADVANCED DEPENDENCY RULES
```

Normal forms provide progressively stronger rules for organizing relational data and reducing unnecessary dependencies and redundancy.

End of Normal Forms

Meptrasoft AI TECHNOLOGIES

4.1 Architecture Overview (Query Flow)

Level: 4 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

When you send a SQL query, the database does not immediately return the answer.

The query passes through several internal components that **parse, optimize, execute, retrieve, and return the result**.

A simplified query flow is:

```text
SQL QUERY
   ↓
PARSER
   ↓
OPTIMIZER
   ↓
EXECUTOR
   ↓
STORAGE / CACHE
   ↓
RESULT
```

Understanding this flow helps explain why database systems can execute the same query in different ways and why some queries perform better than others.

> *A SQL query is processed by multiple database components before the final result reaches the client.*

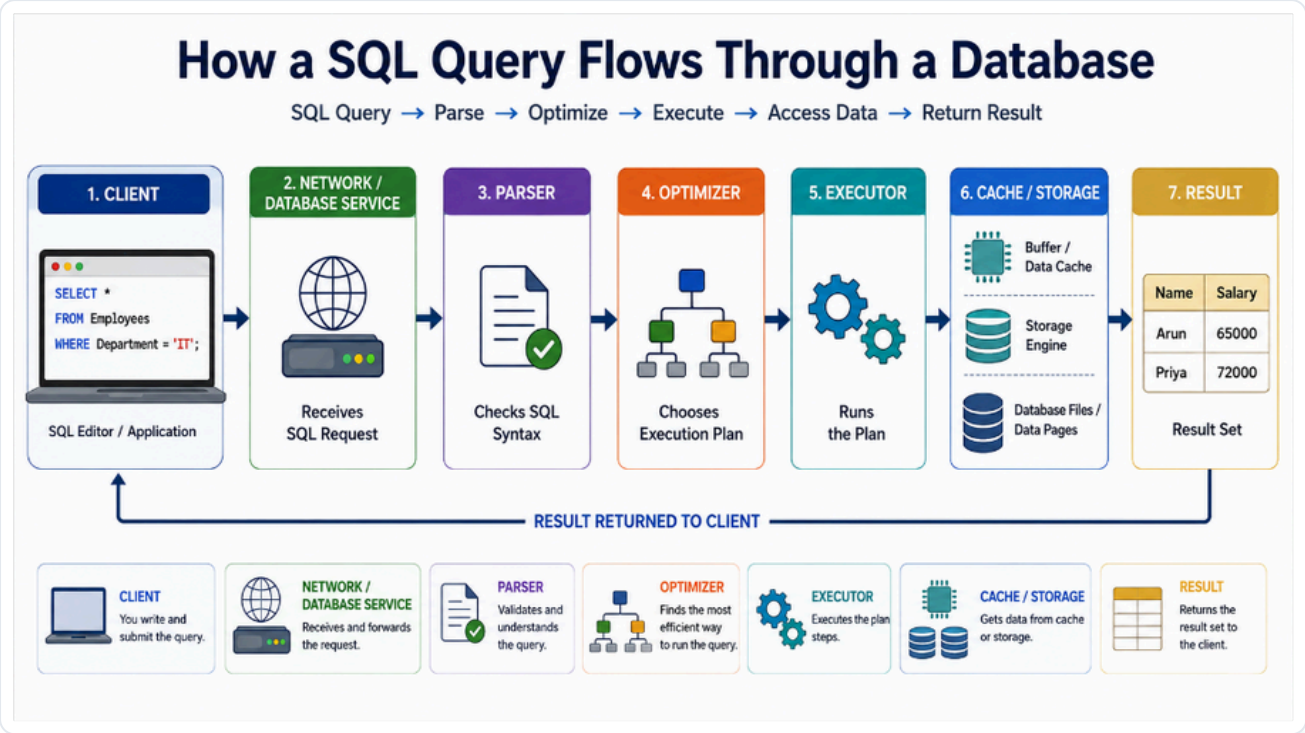


Figure 1: A SQL query passes through communication, parsing, optimization, execution, and data-access components before the result is returned.

Step 1: Client Sends the Query

The process starts when a client application sends a SQL statement to the database.

The client could be:

- SQL command-line tool
- Database management application
- Backend application
- Web application

For example:

```sql
SELECT Name, Salary
FROM Employees
WHERE Department = 'IT';
```

The database receives this request through its communication interface.

Step 2: Query Parser

The **query parser** checks whether the SQL statement follows the database's syntax and can be interpreted.

For example:

```sql
SELECT Name
FROM Employees;
```

is valid SQL syntax.

But:

```sql
SELEC Name
FROM Employees;
```

contains a syntax error.

The parser helps identify such problems before execution continues.

The parser may also create an internal representation of the query that later stages can work with.

> *Parser = Understand and validate the SQL statement.*

Step 3: Query Optimizer

After parsing, the database optimizer determines an efficient way to execute the query.

Suppose the query is:

```SQL
SELECT *
FROM Employees
WHERE Department = 'IT';
```

There may be several possible ways to find the required rows.

The optimizer considers factors such as:

- Available indexes
- Table statistics
- Join methods
- Estimated number of rows
- Available access paths

It then produces an **execution plan**.

> *Optimizer = Decide how the query should be executed efficiently.*

Step 4: Query Executor

The **query executor** follows the selected execution plan.

It performs operations such as:

- Reading rows
- Filtering rows
- Joining tables
- Sorting results
- Calculating values

For example:

```
TEXT

EXECUTION PLAN
      ↓
READ DATA
      ↓
FILTER
      ↓
RETURN MATCHING ROWS
```

Copy

The executor works with the storage and memory components required to obtain the data.

> ***Executor = Run the selected execution plan.***

Step 5: Storage Engine and Data Access

The executor needs access to the actual data.

The database's storage components handle how data is read from and written to persistent storage.

Depending on the database system, this can involve:

- Data pages
- Indexes
- Buffer/cache memory
- Database files

For example:

```text
QUERY EXECUTOR
      ↓
STORAGE ENGINE
      ↓
INDEX / DATA PAGES
      ↓
DATABASE STORAGE
```

The exact architecture varies between database products, so the names and responsibilities of internal components are not identical in every RDBMS.

Step 6: Caching

Reading from disk or other persistent storage is generally slower than reading data already available in memory.

Database systems therefore use memory caches, such as **buffer pools**, to keep frequently needed database pages available in memory.

A simplified idea is:

```text
QUERY
  ↓
CHECK MEMORY CACHE
  ↓
DATA FOUND?
 ↙       ↘
YES       NO
 ↓         ↓
READ      READ FROM STORAGE
CACHE         ↓
          LOAD INTO CACHE
```

Caching can reduce repeated physical I/O and improve query performance.

Step 7: Transaction and Write Management

For queries that modify data, the database must also manage transactional and recovery-related work.

For example:

```sql
UPDATE Accounts
SET Balance = Balance - 1000
WHERE Account_ID = 1;
```

The database may need to coordinate:

- Transaction state
- Locks or other concurrency mechanisms
- Changes to data pages
- Transaction logging
- Recovery information

This helps maintain reliable database operations.

Step 8: Result Returns to the Client

After the query has been executed, the result is sent back to the client.

The overall simplified flow becomes:

```text
CLIENT
  ↓
NETWORK / DATABASE SERVICE
  ↓
PARSER
  ↓
OPTIMIZER
  ↓
EXECUTOR
  ↓
CACHE / STORAGE
  ↓
RESULT
  ↓
CLIENT
```

For example:

```sql
SELECT Name, Salary
FROM Employees
WHERE Department = 'IT';
```

may return:

Name	Salary
Arun	65000
Priya	72000

Query Flow: Read vs Write

The internal work is not exactly the same for every SQL statement.

Read Query

For a `SELECT`:

```
TEXT

SELECT
 ↓
PARSE
 ↓
OPTIMIZE
 ↓
EXECUTE
 ↓
READ CACHE / STORAGE
 ↓
RETURN RESULT
```

Write Query

For an `INSERT`, `UPDATE`, or `DELETE`, the database must also manage transaction and recovery-related operations.

```
TEXT

INSERT / UPDATE / DELETE
          ↓
       PARSE
          ↓
      OPTIMIZE
          ↓
       EXECUTE
          ↓
TRANSACTION / LOGGING
          ↓
      DATA UPDATE
```

This is why database internals are more than simply **"SQL goes directly to the table."**

A Simple Real-World Example

Suppose an employee application asks:

> *"Show all employees from the IT department."*

The application sends:

```sql
SELECT Name, Salary
FROM Employees
WHERE Department = 'IT';
```

The database roughly performs:

```text
1. Receive SQL
       ↓
2. Parse SQL
       ↓
3. Build an execution strategy
       ↓
4. Execute the plan
       ↓
5. Find required data
       ↓
6. Return result
```

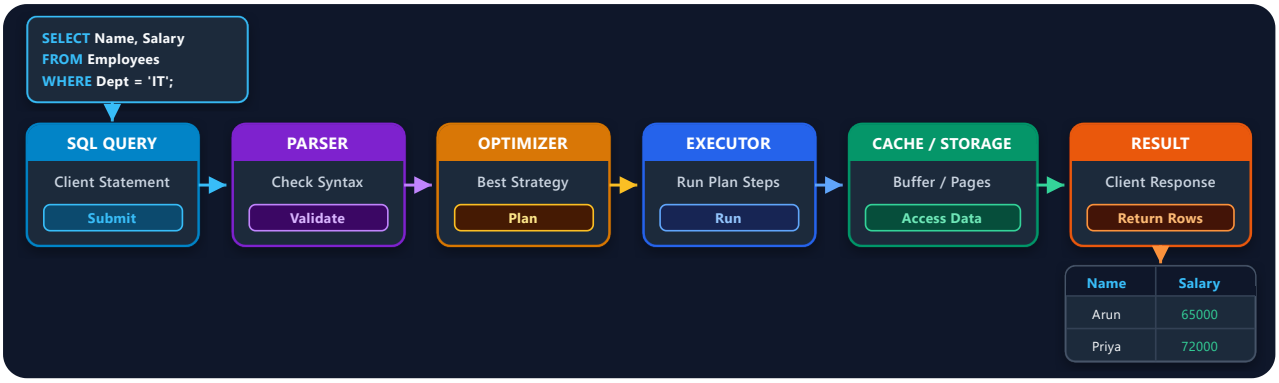


Figure 2: A simplified SQL query flow from the submitted statement to the final result.

Why Query Flow Matters for Performance

Understanding the query flow helps explain why two SQL queries that return the same result can have different performance.

For example, suppose an employee table contains millions of rows.

A query that can use an appropriate index may access fewer data pages than a query that requires a full table scan.

The optimizer considers available access paths and chooses an execution plan based on the database's statistics and cost estimates.

This is the foundation for topics such as:

- Indexes
- Execution plans
- Query optimization
- Join strategies
- Query performance

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
SQL directly reads the disk and returns the result	The database processes the query through multiple internal components
The optimizer executes the query	The optimizer selects an execution strategy; the executor runs it
Every query always reads from disk	Data may already be available in memory/cache
Every database has exactly the same internal architecture	Internal components and terminology differ between database systems
Indexes always make every query faster	Indexes can improve some access patterns but also add storage and write-maintenance costs
Query optimization changes the requested result	The optimizer should choose an execution strategy that produces the requested result according to database semantics

Placement Quick Points

```text
CLIENT
→ SENDS SQL

PARSER
→ CHECKS / INTERPRETS SQL

OPTIMIZER
→ CHOOSES EXECUTION PLAN

EXECUTOR
→ RUNS THE PLAN

CACHE / STORAGE
→ PROVIDES DATA

RESULT
→ RETURNS TO CLIENT
```

Copy

- A SQL query passes through multiple internal stages before returning a result.
- The **parser** validates and interprets the SQL statement.
- The **optimizer** selects an execution strategy.
- The **executor** runs the selected plan.
- Cache and storage components provide the required data.
- Read and write queries can involve different internal operations.
- Database products implement these components differently.

Interview Questions

Q1. What happens when a SQL query is submitted?

A simplified process is:

```
text Receive → Parse → Optimize → Execute → Access Data → Return Result
```

Q2. What does the query parser do?

The parser checks and interprets the SQL statement and identifies syntax or other problems that prevent the query from being processed.

Q3. What is the role of the query optimizer?

The optimizer chooses an efficient execution strategy, usually represented as an execution plan.

Q4. What does the query executor do?

The executor runs the selected execution plan and performs operations needed to produce the result.

Q5. Why is caching useful?

Caching allows frequently needed database pages to be served from memory instead of repeatedly reading them from persistent storage, which can reduce I/O work.

Q6. Does every database use exactly the same query architecture?

No. The overall concepts are similar, but internal architecture, terminology, and implementation vary across database products.

Q7. Why is query flow important for performance?

It helps explain how factors such as indexes, execution plans, joins, memory, and storage access influence query execution time.

Q8. What is an execution plan?

An execution plan is the strategy chosen by the database system for executing a SQL query.

Practice & Hands-On Exercises

Beginner Practice

1. Write the basic stages of SQL query processing in order.
2. What is the role of the parser?
3. What is the role of the optimizer?
4. What is the role of the executor?
5. Why can a database read data from memory instead of storage?

Think and Answer

1. Why might two queries that return the same result have different performance?
2. What could happen when a query needs to read millions of rows?
3. Why are indexes important to query processing?
4. Why can database architecture differ between PostgreSQL, MySQL, SQL Server, and Oracle?

Query Flow Practice

For this query:

```text
SELECT Name
FROM Employees
WHERE Department = 'IT';
```

Copy

1. What does the parser do?
2. What does the optimizer decide?
3. What does the executor perform?
4. Where might the database obtain the required data?
5. What is finally returned to the application?

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```SQL
SQL QUERY
    ↓
PARSER
    ↓
OPTIMIZER
    ↓
EXECUTOR
    ↓
CACHE / STORAGE
    ↓
RESULT
```

> ***A** SQL query passes through parsing, optimization, execution, and data-access stages before the database returns the final result.*

End of Architecture Overview (Query Flow)

4.2 Compute Engine Layer

Level: 4 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

The **Compute Engine** is the part of the database system responsible for **processing SQL queries**.

When a SQL query arrives, the database must:

1. Understand the query.
2. Check whether it is valid.
3. Decide how to execute it efficiently.
4. Execute the selected plan.
5. Return the result.

A simplified flow is:

```text
SQL QUERY
   ↓
PARSER
   ↓
OPTIMIZER
   ↓
EXECUTOR
   ↓
RESULT
```

> *The Compute Engine turns a SQL request into an execution plan and runs that plan to produce a result.*

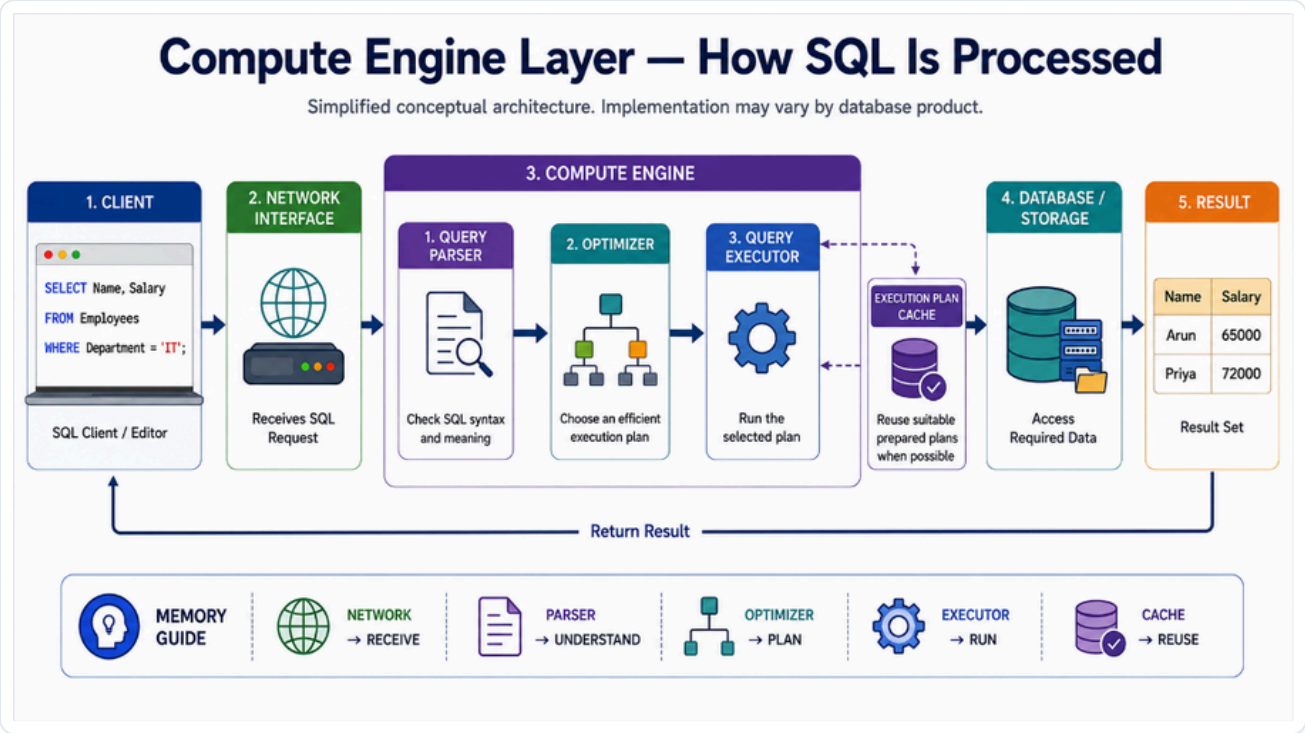


Figure 1: The Compute Engine processes SQL through parsing, optimization, and execution before the final result is returned.

Network Interface — Communication Gateway

The **Network Interface** is the communication entry point between the client and the database system.

It receives SQL requests, manages communication/session-related work, and sends results back to the client.

For example:

It acts as the path through which requests and responses travel.

> *Network Interface = Communication between the client and database system.*

Query Parser

The **Query Parser** examines the SQL statement before it can be executed.

Two important checks are:

1. Syntactic Analysis

The parser checks whether the SQL follows the required grammar and structure.

For example:

```SQL
SELECT Name
FROM Employees;
```

has valid SQL structure.

But:

```SQL
SELEC Name
FROM Employees;
```

contains a syntax error.

2. Semantic Analysis

The database then checks whether the query makes sense in the current database context.

For example:

```sql
SELECT EmployeeName
FROM Employees;
```

If `EmployeeName` does not exist in the table, the query cannot be executed correctly.

Other checks can include:

- Does the referenced table exist?
- Does the referenced column exist?
- Is the requested operation allowed?

> ***Parser = Check whether the SQL is correctly written and meaningful for the database.***

Optimizer

After the query has been parsed, the **Optimizer** determines how the database should execute it.

A query can often be executed in more than one way.

For example, the database may choose between:

```text
Full Table Scan
       OR
Index-Based Access
```

The optimizer considers available information such as:

- Indexes
- Table statistics
- Estimated number of rows
- Possible join methods
- Available access paths

It then selects an **execution plan** based on estimated cost.

> *Optimizer = Choose an efficient way to execute the query.*

Query Executor

The **Query Executor** runs the execution plan selected by the optimizer.

It may perform operations such as:

- Reading data
- Filtering rows
- Joining tables
- Sorting data
- Calculating values

For example:

```text
EXECUTION PLAN
      ↓
READ DATA
      ↓
FILTER
      ↓
PRODUCE RESULT
```

The executor works with the database's data-access and storage components to obtain the required data.

> *Executor = Execute the selected plan and produce the result.*

Execution Plan Cache

Database systems may keep previously prepared execution plans in memory or other internal caches.

When a suitable cached plan can be reused, the database may avoid repeating some planning work.

A simplified idea is:

```text
SQL QUERY
    ↓
PLAN CACHE
    ↓
PLAN AVAILABLE?
   ↙      ↘
 YES       NO
  ↓         ↓
REUSE     CREATE PLAN
```

Plan caching can reduce repeated parsing/planning overhead in appropriate workloads.

> ***Execution Plan Cache = Reuse suitable previously prepared plans when possible.***

Note: Exact plan-cache behavior varies between database products and configurations.

Complete Compute Engine Flow

Consider this query:

```sql
SELECT Name, Salary
FROM Employees
WHERE Department = 'IT';
```

The simplified processing flow is:

```text
1. CLIENT SENDS SQL
          ↓
2. NETWORK INTERFACE RECEIVES IT
          ↓
3. PARSER CHECKS THE QUERY
          ↓
4. OPTIMIZER CHOOSES A PLAN
          ↓
5. EXECUTOR RUNS THE PLAN
          ↓
6. DATA IS ACCESSED
          ↓
7. RESULT IS RETURNED
```

This is the main idea students should remember.

A Simple Real-World Example

Imagine an employee application asking:

> *"Show the names and salaries of employees in the IT department."*

The application sends:

```sql
SELECT Name, Salary
FROM Employees
WHERE Department = 'IT';
```

The database roughly performs:

```text
Request
  ↓
Parse
  ↓
Optimize
  ↓
Execute
  ↓
Return Result
```

The user does not need to know whether the database used an index, a table scan, or another access strategy.

The database chooses the execution approach internally.

How the Components Work Together

Component	Main Responsibility
Network Interface	Receives requests and returns responses
Query Parser	Checks and interprets SQL
Optimizer	Chooses an efficient execution plan
Query Executor	Runs the execution plan
Execution Plan Cache	Reuses suitable prepared plans when possible

Memory Trick

```text
NETWORK   → RECEIVE
PARSER    → UNDERSTAND
OPTIMIZER → PLAN
EXECUTOR  → RUN
CACHE     → REUSE
```

Why the Compute Engine Matters

The Compute Engine is important because a database may receive thousands or millions of SQL requests.

Efficient query processing can reduce:

- Unnecessary data access
- CPU work

- Memory usage
- Query response time

This is why concepts such as **indexes, execution plans, joins, statistics, and query optimization** become important when studying database performance.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
The parser executes the query	The parser checks and interprets the query; the executor runs the plan
The optimizer retrieves the final rows	The optimizer selects an execution strategy
The executor decides the best plan	The optimizer chooses the plan; the executor runs it
Every query must read directly from disk	Data may already be available through memory/cache
Every database has exactly the same Compute Engine design	Internal implementation differs between database products
A cached plan is always reused	Reuse depends on the database system, query, configuration, and whether a suitable plan is available
A faster-looking plan is always faster	The optimizer relies on estimates and database statistics; actual performance depends on the workload and environment

Placement Quick Points

```text
NETWORK INTERFACE
→ RECEIVES SQL REQUEST

PARSER
→ CHECKS / INTERPRETS SQL

OPTIMIZER
→ CHOOSES EXECUTION PLAN

EXECUTOR
→ RUNS EXECUTION PLAN

PLAN CACHE
→ REUSES SUITABLE PLANS WHEN POSSIBLE
```

Copy

- The Compute Engine is the query-processing part of the database system.
- The Network Interface handles communication with clients.
- The Parser checks SQL syntax and meaning.
- The Optimizer selects an execution strategy.
- The Executor runs the selected execution plan.
- Plan caching can reduce repeated planning work when a suitable plan can be reused.
- Exact implementation details vary across database products.

Interview Questions

Q1. What is the Compute Engine?

> The Compute Engine is the query-processing layer responsible for parsing SQL, selecting an execution strategy, and executing the query.

Q2. What does the Query Parser do?

It checks the SQL statement's syntax and interprets it in the context of the database, including references to database objects and applicable permissions or rules.

Q3. What is the role of the Query Optimizer?

The optimizer evaluates possible execution strategies and chooses an execution plan based on estimated cost and available information.

Q4. What does the Query Executor do?

The executor runs the selected execution plan and performs the operations needed to produce the result.

Q5. What is an execution plan?

An execution plan is the database's selected strategy for executing a SQL query.

Q6. What is an execution plan cache?

It is an internal mechanism that can store prepared execution plans so suitable plans may be reused instead of repeating some planning work.

Q7. What is the difference between the optimizer and executor?

```
text Optimizer → Decides HOW to execute Executor → Executes that decision
```

Q8. Why is query optimization important?

It can reduce unnecessary data access and computational work, which can improve query performance.

Q9. Does every database have the same Compute Engine architecture?

No. Database products use different internal architectures and terminology, although the general concepts of parsing, planning/optimization, and execution are common.

Practice & Hands-On Exercises

Beginner Practice

1. What is the purpose of the Compute Engine?
2. What does the Query Parser do?
3. What does the Optimizer do?
4. What does the Query Executor do?
5. What is an execution plan?

Think and Answer

1. Why might a database choose an index instead of scanning an entire table?
2. Why is the optimizer important when a query can be executed in multiple ways?
3. Why can caching improve repeated query performance?
4. What is the difference between parsing, optimizing, and executing?

Query Flow Practice

For:

```
CODE

SELECT Name
FROM Employees
WHERE Department = 'IT';
```

1. Which component checks the SQL?
2. Which component chooses the execution plan?
3. Which component runs the plan?
4. Where might the required data be obtained from?
5. What is returned to the client?

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE

Copy

```
SQL QUERY
    ↓
NETWORK
    ↓
PARSER
    ↓
OPTIMIZER
    ↓
EXECUTOR
    ↓
DATA ACCESS
    ↓
RESULT
```

> *The Compute Engine processes SQL by parsing the query, selecting an execution plan, and executing that plan to produce the required result.*

End of Compute Engine Layer

4.3 Storage Engine Layer

Level: 4 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

The **Storage Engine** is the database layer responsible for **managing access to stored data**.

A simple way to understand the difference is:

> ***Compute Engine decides how to process a query. Storage Engine handles how the required data is accessed or changed.***

For example:

```
TEXT

SQL QUERY
    ↓
COMPUTE ENGINE
    ↓
STORAGE ENGINE
    ↓
DATA
```

Copy

The Storage Engine works with database pages, indexes, memory buffers, transactions, and persistent storage to read or modify data.

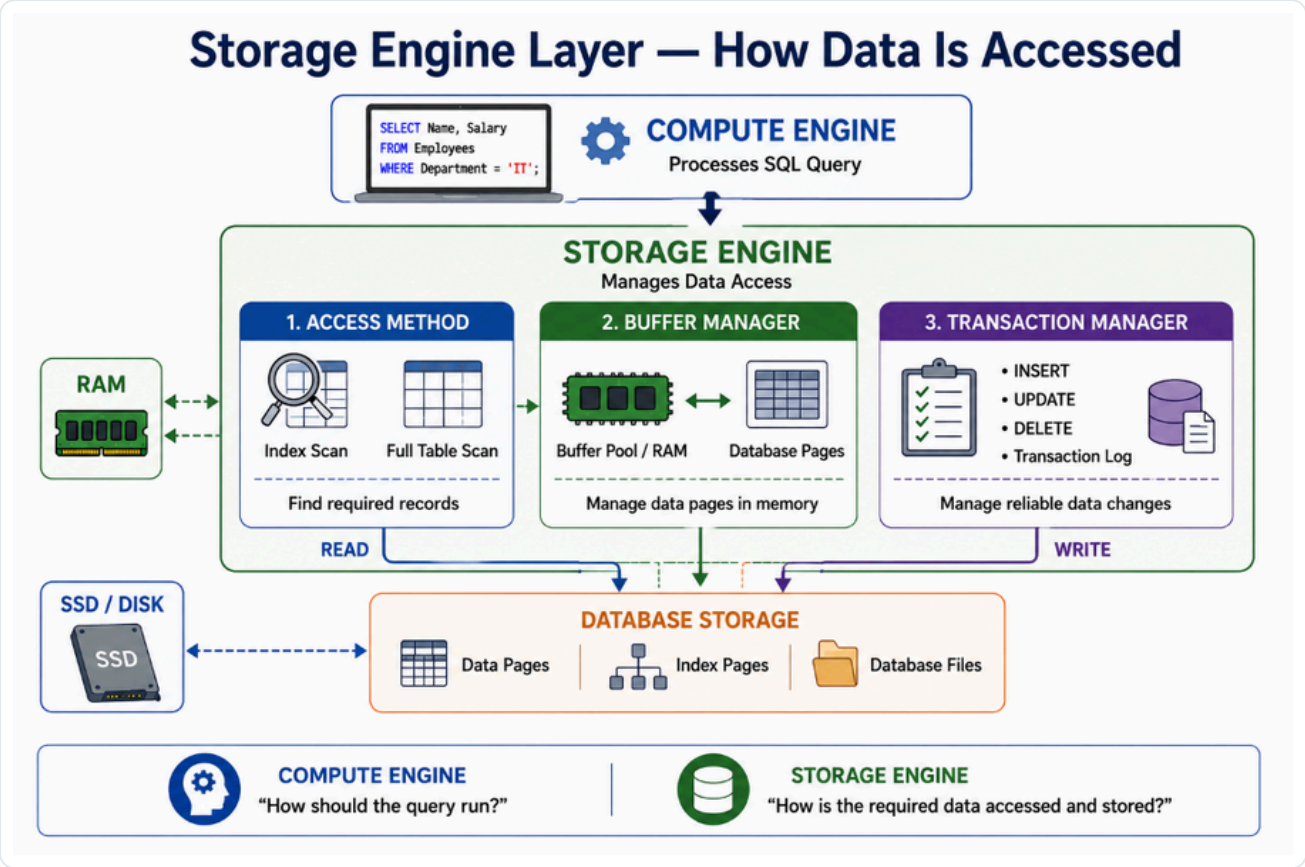


Figure 1: The Storage Engine manages how database data is accessed, cached, modified, and persisted.

Access Method

The **Access Method** determines how the database obtains or modifies the required records.

Two common read strategies are:

Index Scan

If a suitable index exists, the database may use it to locate matching rows more efficiently.

```text
QUERY
  ↓
INDEX
  ↓
MATCHING ROWS
```

Full Table Scan

The database can read the table's data pages and examine rows to find matches.

```text
QUERY
  ↓
READ TABLE PAGES
  ↓
CHECK ROWS
  ↓
MATCHING ROWS
```

For example:

```sql
SELECT *
FROM Employees
WHERE Employee_ID = 101;
```

If a suitable index exists on `Employee_ID`, the optimizer may choose an index-based access path.

> *Access Method = Decide how the required data is accessed.*

The exact access methods available depend on the database system and execution plan.

Buffer Manager

The **Buffer Manager** manages database pages held in memory.

Database data is stored persistently, but reading every required page from storage repeatedly can be expensive.

The buffer manager helps by keeping frequently needed pages in **RAM**.

A simplified flow is:

```text
QUERY NEEDS DATA
      ↓
CHECK BUFFER
      ↓
PAGE AVAILABLE?
   ↙         ↘
 YES          NO
  ↓            ↓
READ RAM    READ STORAGE
               ↓
          LOAD INTO BUFFER
```

This is one reason repeated access to frequently used data can be faster.

Simple Analogy

Think of a student studying from a book:

- **Disk / storage** → the bookshelf
- **RAM / buffer** → the pages already on the desk

Getting a page from the desk is quicker than walking to the bookshelf every time.

> *Buffer Manager = Manages database pages in memory to reduce unnecessary storage I/O.*

Transaction Manager

The **Transaction Manager** coordinates operations that change database state, such as:

```
TEXT

INSERT
UPDATE
DELETE
```

Copy

It works with transaction, locking/concurrency, logging, and recovery mechanisms to help maintain reliable changes.

For example:

```sql
BEGIN;

UPDATE Accounts
SET Balance = Balance - 1000
WHERE Account_ID = 1;

COMMIT;
```

Copy

The transaction manager is involved in ensuring the change is handled according to the database's transaction rules.

> *Transaction Manager = Coordinates database changes as part of reliable transactions.*

Read and Write Paths

At a simplified level, database activity can be thought of as having different paths.

Read Example

For a `SELECT` statement:

```text
SELECT
  ↓
EXECUTION PLAN
  ↓
ACCESS METHOD
  ↓
BUFFER / STORAGE
  ↓
RESULT
```

Write Example

For an `UPDATE`:

```text
UPDATE
  ↓
EXECUTION PLAN
  ↓
ACCESS METHOD
  ↓
TRANSACTION MANAGEMENT
  ↓
DATA PAGES / LOGGING
  ↓
COMMIT
```

Important Note

These paths are **conceptual**, not strict rules.

A `SELECT` can still involve transaction-related behavior, and write operations also use buffers and storage pages. Database systems coordinate these components together rather than treating reads and writes as completely separate pipelines.

Storage Engine and Persistent Storage

Ultimately, database data must be stored somewhere persistently.

A simplified model is:

```text
STORAGE ENGINE
      ↓
DATA / INDEX PAGES
      ↓
DATABASE FILES
      ↓
SSD / DISK
```

The Storage Engine abstracts many of these physical details from the application.

An application does not normally need to know which physical disk location contains a particular employee record.

Example: Finding an Employee

Consider:

```sql
SELECT Name, Department
FROM Employees
WHERE Employee_ID = 101;
```

A simplified flow might be:

```text
SQL QUERY
    ↓
COMPUTE ENGINE
    ↓
EXECUTION PLAN
    ↓
ACCESS METHOD
    ↓
CHECK BUFFER
    ↓
INDEX / DATA PAGES
    ↓
EMPLOYEE RECORD
    ↓
RESULT
```

If the required page is already in memory, the database may avoid reading it again from persistent storage.

If it is not available, the required page may be loaded into the buffer.

Example: Updating an Account

Consider:

```sql
BEGIN;

UPDATE Accounts
SET Balance = Balance - 500
WHERE Account_ID = 1;

COMMIT;
```

The database must not only find the correct account record. It must also manage the change as part of a transaction.

Conceptually:

```text
UPDATE REQUEST
      ↓
ACCESS ACCOUNT DATA
      ↓
TRANSACTION MANAGEMENT
      ↓
MODIFY DATA PAGE
      ↓
LOG / RECOVERY WORK
      ↓
COMMIT
```

The exact internal sequence varies by database system.

Storage Engine vs Compute Engine

These two layers are easy to confuse.

Compute Engine	Storage Engine
Processes SQL queries	Manages data access and storage
Parses and optimizes queries	Reads and modifies data
Produces execution plans	Works with indexes, pages, buffers, and persistent storage
Focuses on query execution logic	Focuses on data management

A simple memory trick:

```text
COMPUTE ENGINE
→ “HOW SHOULD THE QUERY RUN?”

STORAGE ENGINE
→ “HOW DO WE ACCESS THE DATA?”
```

Copy

Why the Storage Engine Matters for Performance

Storage access can be expensive compared with working with data already in memory.

Performance is therefore affected by things such as:

- Index usage
- Data page access
- Buffer/cache efficiency
- Storage I/O
- Query access patterns

For example, an appropriate index may reduce the amount of data the database needs to examine.

However, indexes are not free. They consume storage and can add work to data modifications.

> *Efficient data access is an important part of database performance.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Storage Engine means only the hard disk	It includes the mechanisms used to access, cache, modify, and persist database data
Buffer Manager stores the permanent database	It manages database pages in memory; persistent copies remain in storage
Every `SELECT` is a full table scan	The optimizer may choose an index or another access method
`SELECT` never uses transaction mechanisms	Reads can participate in transactions and isolation depending on the database
`UPDATE` goes directly to disk	Database systems use buffers, transaction management, logging, and storage mechanisms
The Access Method always means an index	Access can use an index, table scan, or other supported access path
All database products have identical storage-engine architecture	Internal implementation differs significantly between products

Placement Quick Points

```text
STORAGE ENGINE
→ MANAGES DATA ACCESS AND STORAGE

ACCESS METHOD
→ INDEX SCAN / TABLE SCAN / OTHER ACCESS PATH

BUFFER MANAGER
→ MANAGES DATABASE PAGES IN MEMORY

TRANSACTION MANAGER
→ COORDINATES DATA-MODIFYING TRANSACTIONS

PERSISTENT STORAGE
→ DATABASE FILES / DATA PAGES
```

Copy

- The Storage Engine manages how data is accessed and changed.
- Access methods determine how the required records are located.
- An index scan and a full table scan are common access strategies.
- The Buffer Manager manages database pages in memory.
- The Transaction Manager coordinates reliable data modifications.
- Database systems use persistent storage for durable data.
- Read and write operations use these components in coordinated ways.
- Exact Storage Engine architecture varies between database products.

Interview Questions

Q1. What is the Storage Engine?

The Storage Engine is the database layer responsible for managing access to stored data, including reading, modifying, caching, and persistence.

Q2. What is an Access Method?

An access method is the mechanism used to locate or access the required database data, such as an index scan or table scan.

Q3. What is the role of the Buffer Manager?

The Buffer Manager manages database pages in memory and loads required pages from persistent storage when they are not already available.

Q4. Why is buffering useful?

Keeping frequently needed database pages in RAM can reduce repeated storage I/O and improve performance.

Q5. What does the Transaction Manager do?

It coordinates data-modifying operations with transaction, concurrency, logging, and recovery mechanisms.

Q6. What is the difference between an index scan and a full table scan?

An **index scan** uses an index to locate relevant rows, while a **full table scan** examines the table's data pages to find matching rows.

Q7. Is every `SELECT` handled only by the Buffer Manager?

No. The Storage Engine coordinates multiple components. A `SELECT` may use an access method, buffers, indexes, and persistent storage depending on the execution plan.

Q8. How does the Storage Engine affect performance?

It affects how efficiently the database accesses data, including index usage, memory buffering, and storage I/O.

Practice & Hands-On Exercises

Beginner Practice

1. What is the purpose of the Storage Engine?
2. What does the Access Method do?
3. What does the Buffer Manager do?
4. What does the Transaction Manager do?
5. What is the difference between RAM and persistent database storage?

Think and Answer

1. Why can reading data from memory be faster than reading it from storage?
2. Why might a database choose an index scan instead of a full table scan?
3. Why do write operations need transaction management?
4. Why are the Storage Engine and Compute Engine separate concepts?

Query Flow Practice

For:

```sql
SELECT Name
FROM Employees
WHERE Employee_ID = 101;
```

1. Which layer processes the SQL query?
2. Which component may choose an index-based access path?
3. Where might the required database page already be stored?
4. What happens if the required page is not currently in the buffer?
5. What is finally returned to the client?

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```TEXT
COMPUTE ENGINE
       ↓
STORAGE ENGINE
       ↓
ACCESS METHOD
   ↙       ↘
BUFFER     TRANSACTION
MANAGER      MANAGER
   ↓           ↓
DATA / INDEX PAGES
       ↓
PERSISTENT STORAGE
```

> *The Storage Engine manages how database data is accessed, cached, modified, and persisted, while the Compute Engine handles the processing of the SQL query itself.*

End of Storage Engine Layer

4.4 Physical Storage (Pages, Extents, Data Files, Database Files)

Level: 4 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

When a database stores data permanently, the data eventually has to be written to **persistent storage** such as an SSD or hard disk.

Database systems organize this stored data into logical and physical units so that data can be accessed and managed efficiently.

A simplified view is:

```text
DATABASE
   ↓
DATA FILES
   ↓
PAGES
   ↓
ROWS / INDEX ENTRIES
   ↓
PHYSICAL STORAGE
```

Copy

Different database products use different storage architectures, terminology, and sizes. The concepts below provide a simplified model that is useful for understanding database internals.

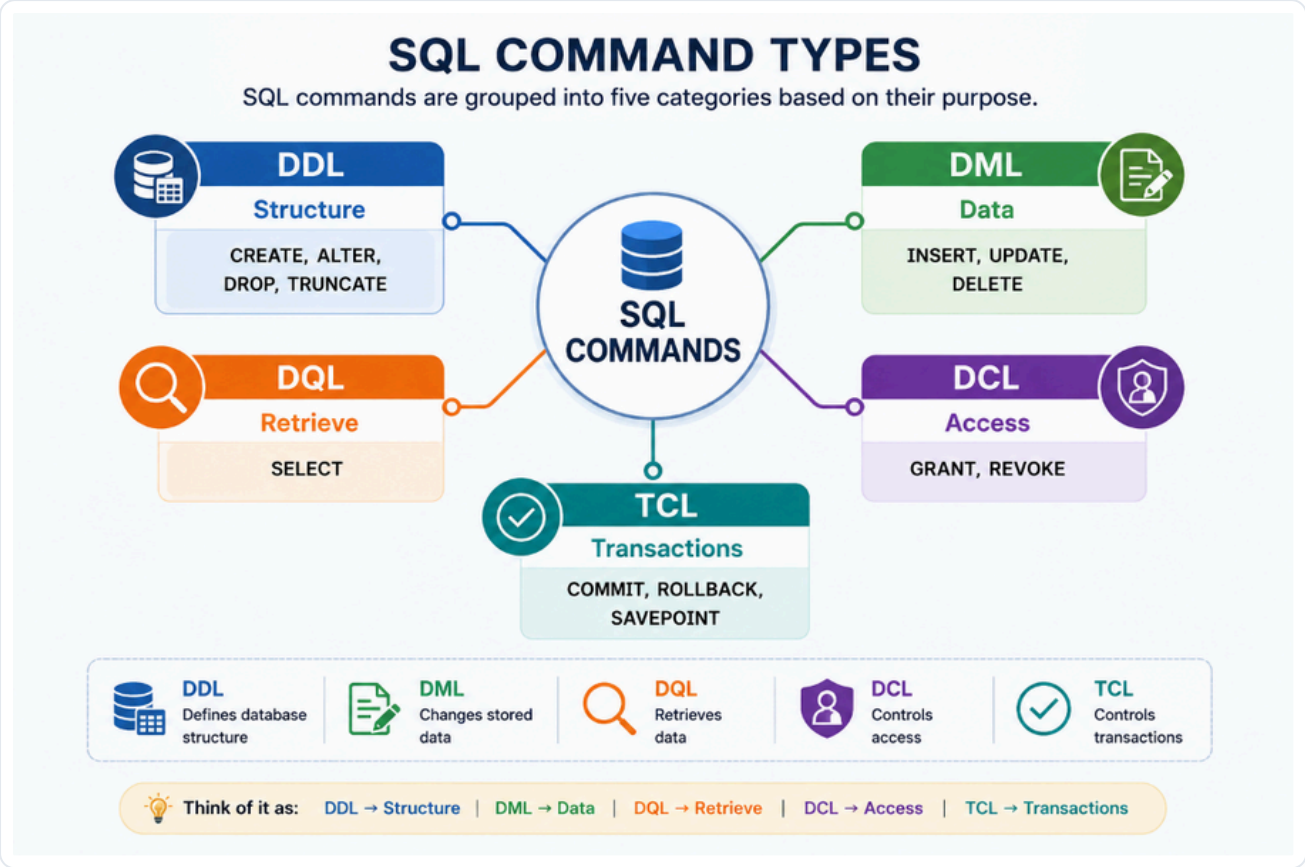


Figure 1: Database systems organize persistent data using structures such as files, pages, and other storage units before it is written to physical storage.

Pages

A **page** is a fixed-size unit of data storage used internally by a database system.

Database systems commonly read and write data in pages rather than processing individual bytes or rows independently.

A page may contain:

- Table rows
- Parts of rows
- Index entries

For example:

```
TEXT

PAGE
┌──────────────────────────┐
│ Row 1                    │
│ Row 2                    │
│ Row 3                    │
│ Index / page information │
└──────────────────────────┘
```

The exact page size depends on the database product.

For example, **SQL Server uses 8 KB data pages**, while other database systems may use different page sizes.

> ***Page = A basic unit of database data storage and I/O.***

Extents

An **extent** is a group of pages allocated together in database systems that use extent-based allocation.

For example, in **SQL Server**, one extent contains **8 pages**, so one extent is 64 KB.

```
TEXT

EXTENT
┌────┬────┬────┬────┬────┬────┬────┬────┐
│ P1 │ P2 │ P3 │ P4 │ P5 │ P6 │ P7 │ P8 │
└────┴────┴────┴────┴────┴────┴────┴────┘
```

The important concept is:

> *Extent = A group of database pages allocated together.*

Important: Extent size and allocation behavior are database-product specific. Do not present `8 pages = 64 KB` as a universal rule for every RDBMS.

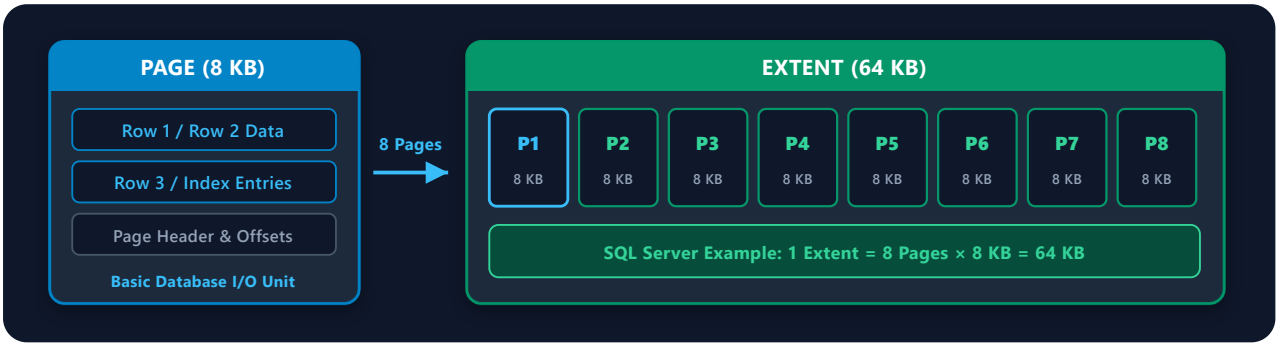


Figure 2: An extent groups multiple pages; the 8-page, 64 KB example shown here applies to SQL Server.

Data Files

A **data file** is a file used by the database system to store database data.

Depending on the database product, data files can contain:

- Table data
- Index data
- Database pages
- Other internally managed structures

Conceptually:

```text
DATA FILE
    ↓
PAGES
    ↓
TABLE / INDEX DATA
```

A database can use one or more data files depending on its architecture and configuration.

Database Files

A database may consist of **multiple physical files**, depending on the database system.

For example, SQL Server databases can use:

- Data files
- Transaction log files

A simplified representation is:

```text
DATABASE
   ├── Data File
   ├── Data File
   └── Log File
```

The exact number and types of files depend on the database product.

> *A database is a logical collection of data managed by the DBMS, while physical files are storage structures used to persist that data.*

Transaction Log

The **transaction log** records changes needed for transaction processing and recovery.

For example, suppose an account balance changes:

```sql
UPDATE Accounts
SET Balance = Balance - 1000
WHERE Account_ID = 1;
```

The database records appropriate information in its transaction log as part of its transaction and recovery mechanisms.

A simplified idea is:

```text
CHANGE REQUEST
      ↓
TRANSACTION LOG
      ↓
DATA PAGES
      ↓
COMMIT
```

The transaction log is important for:

- Recovery after failures
- Maintaining transaction durability
- Replaying or recovering committed changes
- Supporting rollback/recovery mechanisms

> ***Transaction Log = A durable record used by the database system to support transaction processing and recovery.***

Important Correction

The transaction log should not be described simply as **"all data modifications are written to the log before being written to disk."**

The exact write ordering and recovery process depend on the database system. The important concept is that transaction logging is a key part of reliable recovery and durability.

How Data Moves Through Physical Storage

Consider this simplified flow:

```text
SQL QUERY
    ↓
COMPUTE ENGINE
    ↓
STORAGE ENGINE
    ↓
DATA ACCESS
    ↓
PAGE IN MEMORY
    ↓
DATA FILE
    ↓
SSD / DISK
```

The database does not usually think of a table as one giant continuous object.

Instead, table and index data are managed internally using pages and other storage structures.

Example: Finding a Row

Suppose we run:

```sql
SELECT Name
FROM Employees
WHERE Employee_ID = 101;
```

A simplified physical-storage path might be:

```text
SQL QUERY
    ↓
EXECUTION PLAN
    ↓
INDEX / TABLE ACCESS
    ↓
REQUIRED PAGE
    ↓
BUFFER MEMORY
    ↓
EMPLOYEE ROW
    ↓
RESULT
```

If the required page is already in memory, the database may not need to read it again from persistent storage.

This connects the physical storage layer with the **Buffer Manager** discussed in the previous topic.

Physical Storage Hierarchy

The concepts can be viewed together as:

```text
DATABASE
   ↓
DATA / LOG FILES
   ↓
PAGES
   ↓
ROWS / INDEX ENTRIES
   ↓
SSD / DISK
```

An extent, where supported by the database system, is a grouping of pages:

```text
EXTENT
   ↓
PAGE
PAGE
PAGE
PAGE
...
```

Important Note

There is **no universal hierarchy such as**:

```text
8 pages = 1 extent
8 extents = 1 data file
8 data files = 1 database
```

The first relationship is a valid **SQL Server example**, but data-file sizes, file counts, and database sizes are configuration-dependent and vary across database products.

Physical Storage and Query Performance

For example:

- Reading fewer pages can reduce I/O.
- A useful index can reduce the number of pages that must be examined.
- Keeping frequently used pages in memory can reduce storage reads.
- Large amounts of random I/O can affect query performance.

This is why concepts such as **pages, indexes, buffers, and I/O** are important when studying database performance.

```text
FEWER PAGES READ
      ↓
LESS I/O
      ↓
POTENTIALLY BETTER PERFORMANCE
```

Copy

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
A page is always 8 KB in every database	Page size depends on the database product
Every database uses 8-page extents	Extent allocation is database-specific
A data file always has a fixed number of extents	Data-file size and allocation depend on configuration and product
A database always has exactly 8 data files	File count is configurable and database-specific
The transaction log is the same as a data file	The transaction log serves transaction and recovery purposes
A database table is stored as one giant block	Database systems manage table and index data using storage units such as pages
The application directly reads database pages from disk	The DBMS/storage subsystem manages physical data access

Placement Quick Points

```text
PAGE
→ BASIC DATABASE STORAGE / I/O UNIT

EXTENT
→ GROUP OF PAGES IN SYSTEMS THAT USE EXTENTS

DATA FILE
→ PHYSICAL FILE USED TO STORE DATABASE DATA

LOG FILE
→ SUPPORTS TRANSACTION PROCESSING AND RECOVERY

PHYSICAL STORAGE
→ SSD / DISK
```

Copy

- A **page** is a basic unit used to store and access database data.
- Page size is database-product specific.
- An **extent** is a group of pages in database systems that use extent-based allocation.
- A **data file** stores database data on persistent storage.
- A **transaction log** supports transaction processing and recovery.
- Database file organization varies between products.
- SQL Server provides a common example of 8 KB pages and 8-page extents.
- Physical storage and I/O can significantly affect database performance.

Interview Questions

Q1. What is a database page?

> A page is a fixed-size internal unit used by a database system to store and access data.

Q2. What is an extent?

An extent is a group of pages allocated together in database systems that use extent-based allocation.

Q3. Is the page size always 8 KB?

No. Page size depends on the database system. **SQL Server uses 8 KB data pages**, but other systems may use different sizes.

Q4. What is a data file?

A data file is a physical file used by a database system to persist database data such as table and index pages.

Q5. What is a transaction log?

A transaction log records information about database changes needed for transaction processing and recovery.

Q6. What is the difference between a data file and a log file?

A **data file** stores database data and structures such as table and index pages. A **log file** records transaction-related information used for recovery and transaction management.

Q7. Why are pages important for database performance?

Database systems perform many read and write operations at the page level. The number of pages that must be accessed affects I/O and therefore can affect query performance.

Q8. Are physical-storage structures identical across all RDBMS products?

No. Storage architecture, page sizes, extents, file structures, and allocation methods vary between database products.

Q9. How are pages related to the Buffer Manager?

The Buffer Manager loads database pages into memory so they can be accessed without repeatedly reading them from persistent storage.

Practice & Hands-On Exercises

Beginner Practice

1. What is a database page?
2. What is an extent?
3. What is a data file?
4. What is the purpose of a transaction log?
5. Why does a database use pages instead of treating the entire table as one large block?

Think and Answer

1. Why can reducing the number of pages read improve query performance?
2. Why is frequently accessed data often kept in memory?
3. Why can't we assume that all RDBMS products use the same page and extent sizes?
4. What is the difference between a data file and a transaction log?

SQL / Database Thinking

1. For this query:

```sql
SELECT Name
FROM Employees
WHERE Employee_ID = 101;
```

Explain the simplified path from the query to the physical data.

1. Why might an index reduce the amount of data the database needs to read?
2. How does the Buffer Manager help reduce repeated storage I/O?

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```TEXT
DATABASE
   ↓
FILES
   ↓
PAGES
   ↓
ROWS / INDEX ENTRIES
   ↓
PHYSICAL STORAGE

EXTENT
→ GROUP OF PAGES
```

Copy

> *Physical storage explains how a database organizes persistent data into internal units such as pages and files, while mechanisms such as buffers and transaction logs help make data access and recovery efficient and reliable.*

End of Physical Storage (Pages, Extents, Data Files, Database Files)

5.1 Overview of SQL Command Types

Level: 5 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

SQL commands are used to work with databases in different ways. For easier learning, they are commonly grouped into five categories:

Category	Full Form	Purpose	Main Commands
DDL	Data Definition Language	Manage database structure	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`
DML	Data Manipulation Language	Change data	`INSERT`, `UPDATE`, `DELETE`
DQL	Data Query Language	Retrieve data	`SELECT`
DCL	Data Control Language	Manage permissions	`GRANT`, `REVOKE`
TCL	Transaction Control Language	Manage transactions	`COMMIT`, `ROLLBACK`, `SAVEPOINT`

> *Think of it as:*
> *DDL → Structure*
> *DML → Data*
> *DQL → Retrieve*
> *DCL → Access*
> *TCL → Transactions*

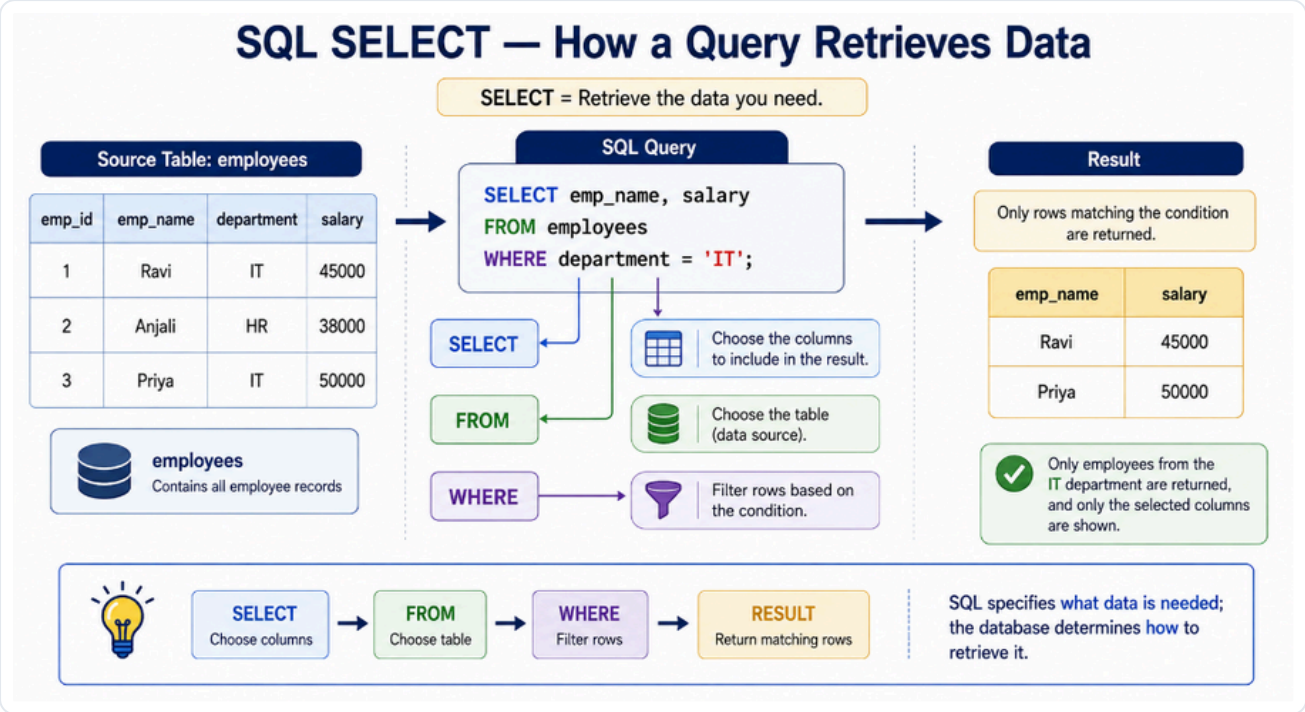


Figure 1: SQL commands are commonly grouped into five categories based on their purpose.

Quick Comparison

```text
DDL → Defines database structure
DML → Changes stored data
DQL → Retrieves data
DCL → Controls access
TCL → Controls transactions
```

The next topics explain each category separately in detail.

Placement Quick Points

- **DDL** → Structure

- **DML** → Data
- **DQL** → Retrieval
- **DCL** → Permissions
- **TCL** → Transactions

Interview Questions

Q1. What are the commonly taught SQL command categories?

> DDL, DML, DQL, DCL, and TCL.

Q2. What is DDL used for?

> Managing database structure.

Q3. What is DML used for?

> Changing data stored in tables.

Q4. What is DQL used for?

> Retrieving data, mainly using `SELECT`.

Q5. What is DCL used for?

> Managing database permissions and access.

Q6. What is TCL used for?

> Managing transactions.

Practice & Hands-On Exercises

1. `CREATE`
2. `INSERT`
3. `SELECT`
4. `UPDATE`
5. `GRANT`
6. `COMMIT`
7. `DROP`
8. `DELETE`
9. `ROLLBACK`

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
DDL → STRUCTURE
DML → DATA
DQL → RETRIEVE
DCL → ACCESS
TCL → TRANSACTION
```

> ***SQL commands are commonly grouped according to what they do with the database.***

End of Overview of SQL Command Types

5.2 DDL — CREATE, DROP, TRUNCATE, ALTER

Level: 5 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

DDL (Data Definition Language) contains SQL commands used to **create, change, and remove database structures** such as tables.

The four commands covered here are:

Command	Main Purpose
CREATE	Create a new database object
ALTER	Change an existing structure
TRUNCATE	Remove all rows but keep the table
DROP	Remove the database object

> ***DDL mainly works with database structure rather than individual row values.***

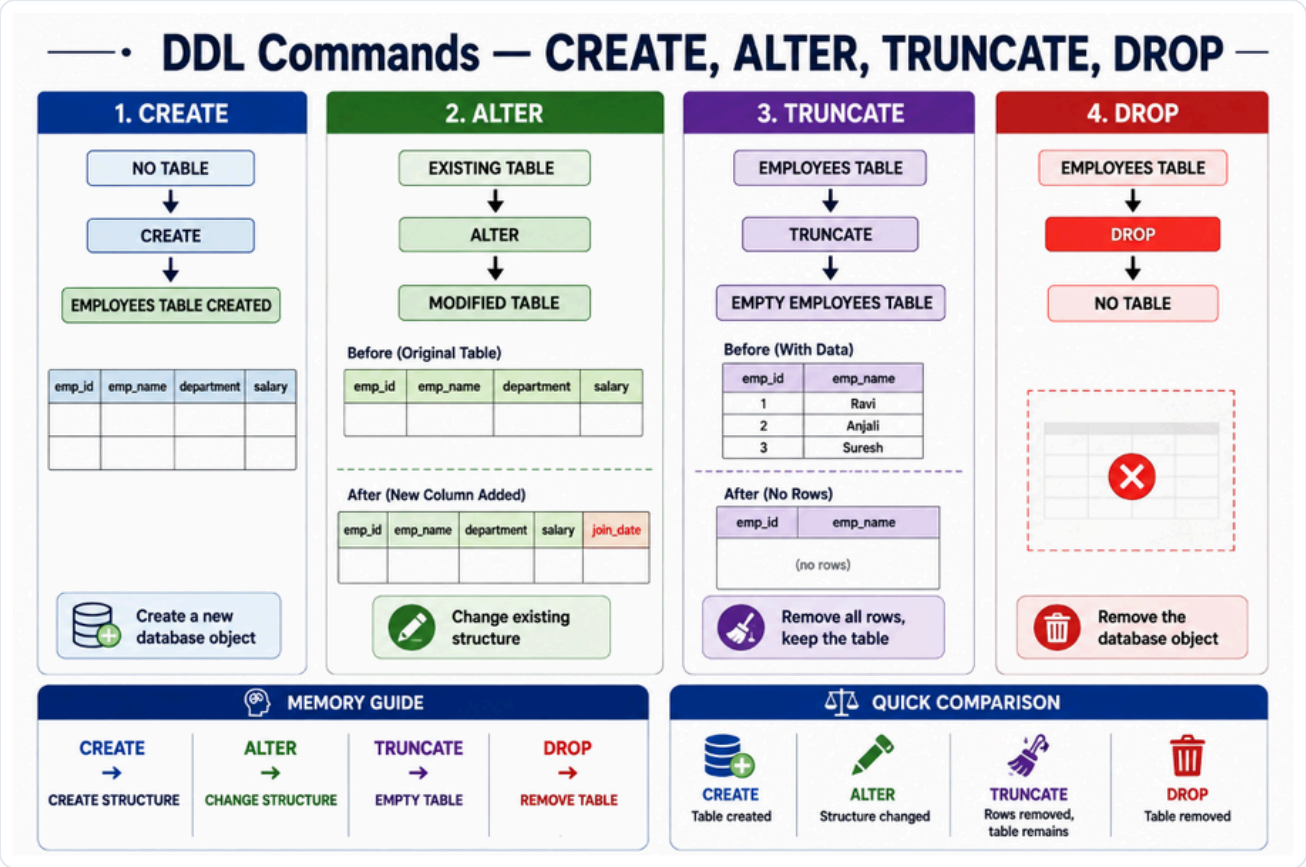


Figure 1: DDL commands manage database structure: CREATE creates objects, ALTER changes structure, TRUNCATE removes all rows while keeping the table, and DROP removes the object.

CREATE

`CREATE` is used to create a new database object.

For example:

```sql
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    department VARCHAR(50),
    salary NUMERIC(10, 2)
);
```

This creates the `employees` table and defines its columns.

> *CREATE → Create structure*

ALTER

`ALTER` is used to change the structure of an existing database object.

Add a Column

```sql
ALTER TABLE employees
ADD COLUMN join_date DATE;
```

Now the table has an additional `join_date` column.

Change a Column Type

For example, in PostgreSQL:

```sql
ALTER TABLE employees
ALTER COLUMN emp_name TYPE VARCHAR(100);
```

Note: `ALTER` syntax can differ between database systems.

> *ALTER → Change structure*

TRUNCATE

`TRUNCATE` removes **all rows** from a table while keeping the table itself.

```sql
TRUNCATE TABLE employees;
```

Before:

emp_id	emp_name
1	Ravi
2	Anjali

After:

emp_id	emp_name
(no rows)	

The table still exists and can be used again.

> ***TRUNCATE → Remove all rows, keep the table***

DROP

`DROP` removes the database object itself.

```sql
DROP TABLE employees;
```

After `DROP`, the table no longer exists.

> ***DROP → Remove the structure***

CREATE vs ALTER vs TRUNCATE vs DROP

Command	Table Exists After Command?	Rows Remain?	Structure Remains?
`CREATE`	Yes	Depends	Creates structure
`ALTER`	Yes	Yes	Modified
`TRUNCATE`	Yes	**No**	**Yes**
`DROP`	**No**	**No**	**No**

Easy Memory Trick

```text
CREATE   → CREATE
ALTER    → CHANGE
TRUNCATE → EMPTY
DROP     → REMOVE
```

Practice Table Setup

SQL

Copy

```sql
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(50),
    department VARCHAR(50),
    salary NUMERIC(10, 2)
);

INSERT INTO employees
(emp_id, emp_name, department, salary)
VALUES
(1, 'Ravi Kumar', 'IT', 45000.00),
(2, 'Anjali Mehta', 'HR', 38000.00),
(3, 'Suresh Rao', 'Finance', 42000.00),
(4, 'Priya Nair', 'IT', 50000.00);
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`DELETE` and `DROP` are the same	`DELETE` removes rows; `DROP` removes the object
`TRUNCATE` removes the table	It removes all rows but keeps the table structure
`ALTER` is used to change row values	`ALTER` changes structure
`CREATE` is used only for tables	It can create different database objects, depending on the DBMS
`DROP` can always be undone with `ROLLBACK`	Behavior depends on the database system; don't assume it can be reversed
All DDL syntax is identical across databases	Some syntax differs between products

Placement Quick Points

```text
CREATE
→ CREATE DATABASE OBJECT

ALTER
→ CHANGE DATABASE STRUCTURE

TRUNCATE
→ REMOVE ALL ROWS
→ KEEP TABLE

DROP
→ REMOVE DATABASE OBJECT
```

Copy

- DDL stands for **Data Definition Language**.
- DDL is mainly used to manage database structure.
- `CREATE` creates an object.
- `ALTER` modifies an existing object's structure.
- `TRUNCATE` removes all rows but keeps the table.
- `DROP` removes the table or other database object.
- Some DDL behavior varies between database products.

Interview Questions

Q1. What is DDL?

> DDL is a group of SQL commands used to create and modify database structures.

Q2. What is the difference between DROP and TRUNCATE?

> `DROP` removes the table itself, while `TRUNCATE` removes all rows but keeps the table structure.

Q3. What is the difference between DELETE and TRUNCATE?

> `DELETE` removes rows and can use a `WHERE` condition. `TRUNCATE` removes all rows from the table and does not use a `WHERE` condition.

Q4. What is ALTER used for?

> `ALTER` is used to modify the structure of an existing database object.

Q5. Give examples of DDL commands.

> `text CREATE ALTER DROP TRUNCATE`

Q6. Does TRUNCATE remove the table?

> No. The table structure remains; its rows are removed.

Q7. Does DROP remove the data?

> Yes. When a table is dropped, the table structure and its stored rows are removed.

Practice & Hands-On Exercises

Using the `employees` table:

1. Add a `join_date` column using `ALTER`:

```
CODE                                                    Copy

ALTER TABLE employees ADD COLUMN join_date DATE;
```

1. Display the table structure:

```
CODE
Copy

SELECT * FROM employees LIMIT 1;
```

1. Remove all rows using `TRUNCATE`:

```
CODE
Copy

TRUNCATE TABLE employees;
```

1. Create the table again using `CREATE`.
2. Finally, remove the table using `DROP`:

```
CODE
Copy

DROP TABLE employees;
```

1. Explain the difference between `TRUNCATE` and `DROP`.
2. Explain the difference between `ALTER` and `UPDATE`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE
Copy

CREATE   → CREATE STRUCTURE
ALTER    → CHANGE STRUCTURE
TRUNCATE → REMOVE ALL ROWS
DROP     → REMOVE STRUCTURE
```

> ***DDL commands manage database structure, with CREATE creating objects, ALTER modifying them, TRUNCATE emptying tables, and DROP removing objects.***

End of DDL — CREATE, DROP, TRUNCATE, ALTER

5.3 DQL — SELECT

Level: 5 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`SELECT` is used to **retrieve data from a database**.

It can return all columns, specific columns, or only rows that match a condition.

```text
SELECT
  ↓
FROM
  ↓
WHERE
  ↓
RESULT
```

> *SELECT = Retrieve the data you need.*

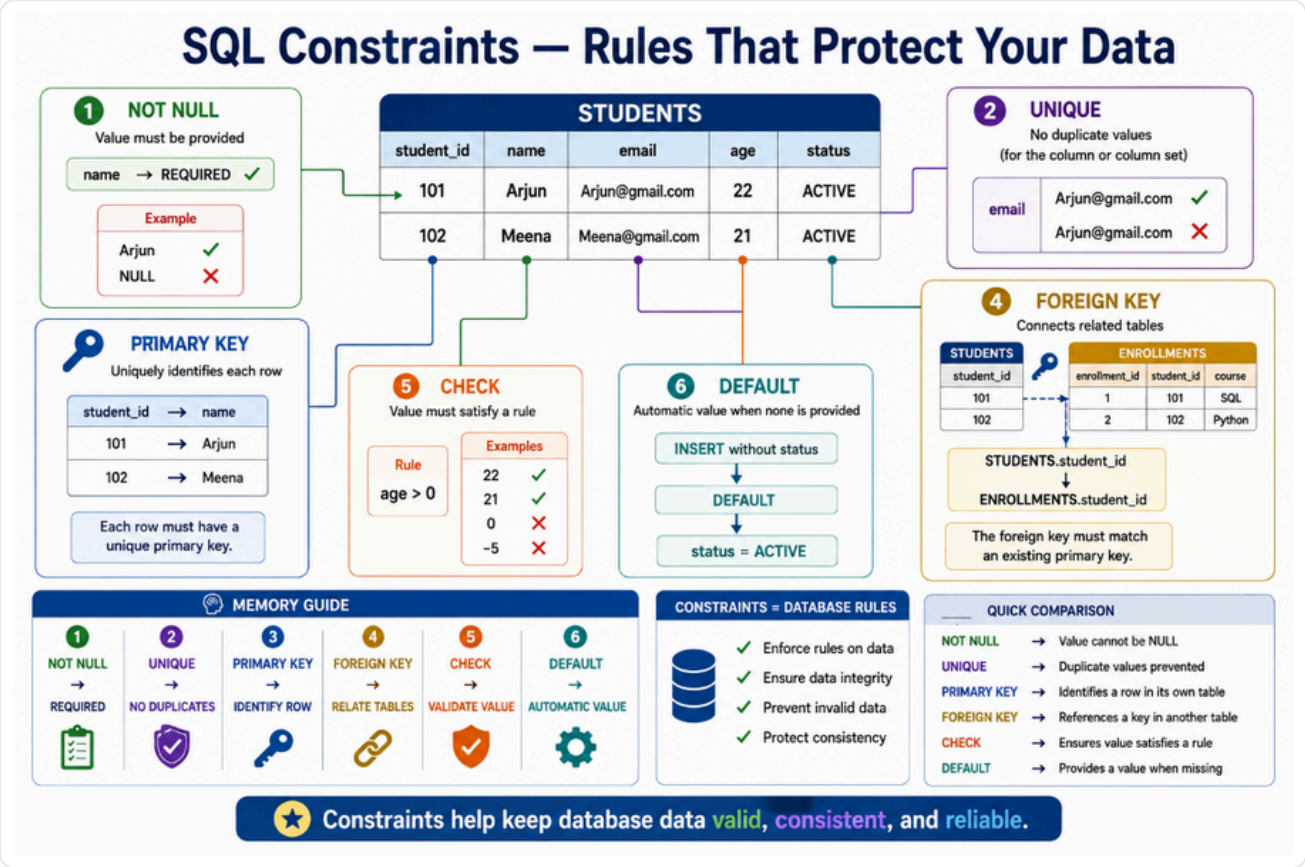


Figure 1: A SELECT query chooses data from a table and can filter the rows returned.

Practice Table

This topic reuses the `employees` table from **5.2 DDL**.

emp_id	emp_name	department	salary
1	Ravi Kumar	IT	45000.00
2	Anjali Mehta	HR	38000.00
3	Suresh Rao	Finance	42000.00
4	Priya Nair	IT	50000.00

Basic SELECT

To retrieve all columns:

```sql
SELECT *
FROM employees;
```

`*` means **all columns**.

SELECT Specific Columns

You can choose only the columns you need:

```sql
SELECT emp_name, department
FROM employees;
```

Result:

emp_name	department
Ravi Kumar	IT
Anjali Mehta	HR
Suresh Rao	Finance
Priya Nair	IT

This is usually clearer than selecting unnecessary columns.

SELECT with WHERE

`WHERE` is used to **filter rows**.

```sql
SELECT *
FROM employees
WHERE department = 'IT';
```

Result:

emp_id	emp_name	department	salary
1	Ravi Kumar	IT	45000.00
4	Priya Nair	IT	50000.00

Simple Idea

```text
FROM   → Which table?
SELECT → Which columns?
WHERE  → Which rows?
```

Logical Query Processing Order

The SQL statement is written in one order, but the database conceptually processes its clauses in a different logical order.

A simplified order is:

```text
FROM / JOIN
      ↓
WHERE
      ↓
GROUP BY
      ↓
HAVING
      ↓
SELECT
      ↓
ORDER BY
      ↓
LIMIT
```

Copy

For example, the database first determines the source rows, applies filtering, performs grouping when required, and then produces the selected output.

> ***The order you write SQL and the logical order used to evaluate it are not always the same.***

Note: The exact physical execution may differ because the optimizer can choose a different execution plan.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`SELECT *` means select all tables	It means all columns from the selected source rows
`WHERE` chooses columns	`WHERE` filters rows
`FROM` filters the result	`FROM` identifies the source table or tables
`SELECT` always returns every row	`WHERE` or other clauses can reduce the returned rows
SQL always executes exactly in the written order	SQL has a logical processing order, and the optimizer chooses the physical execution plan
`SELECT` changes stored data	`SELECT` normally retrieves data without modifying it

Placement Quick Points

```text
SELECT
→ CHOOSE COLUMNS

FROM
→ CHOOSE SOURCE

WHERE
→ FILTER ROWS
```

- `SELECT` retrieves data.
- `FROM` identifies the source table or tables.

- `WHERE` filters rows.
- `*` means all selected columns.
- You can select only the columns needed for the result.
- SQL has a logical query-processing order that differs from the written order.

Interview Questions

Q1. What is SELECT?

`SELECT` is used to retrieve data from a database.

Q2. What does `SELECT *` mean?

It requests all columns from the selected source.

Q3. What is the purpose of FROM?

`FROM` identifies the table or other data source from which the query retrieves data.

Q4. What is the purpose of WHERE?

`WHERE` filters rows based on a condition.

Q5. What is the difference between SELECT and WHERE?

```
text SELECT → Which columns? WHERE → Which rows?
```

Q6. What is the logical order of processing a query?

A simplified order is:

```
text FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT
```

Q7. Does SELECT modify data?

Normally, no. `SELECT` retrieves data without changing the stored rows.

Practice & Hands-On Exercises

Using the `employees` table:

1. Display all columns:

```
CODE
Copy
SELECT * FROM employees;
```

1. Display only `emp_name` and `salary`:

```
CODE
Copy
SELECT emp_name, salary FROM employees;
```

1. Display employees from the `IT` department:

```
CODE
Copy
SELECT * FROM employees WHERE department = 'IT';
```

1. Display employees whose salary is greater than `40000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary > 40000;
```

1. Display only `emp_name` for employees in `HR`:

```
CODE
Copy
SELECT emp_name FROM employees WHERE department = 'HR';
```

1. Explain the difference between `SELECT *` and selecting specific columns.
2. Identify the roles of `SELECT`, `FROM`, and `WHERE` in a query.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE

Copy

```
SELECT → COLUMNS
FROM   → SOURCE
WHERE  → ROWS TO KEEP
```

> ***SELECT is used to retrieve data, while FROM identifies the source and WHERE filters the rows.***

End of DQL — SELECT

5.4 DCL — GRANT, REVOKE

Level: 5 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

DCL (Data Control Language) is commonly used for managing **database permissions and access**.

The two main commands are:

Command	Purpose
`GRANT`	Give permissions
`REVOKE`	Remove permissions

> ***GRANT → Give access***
> ***REVOKE → Take back access***

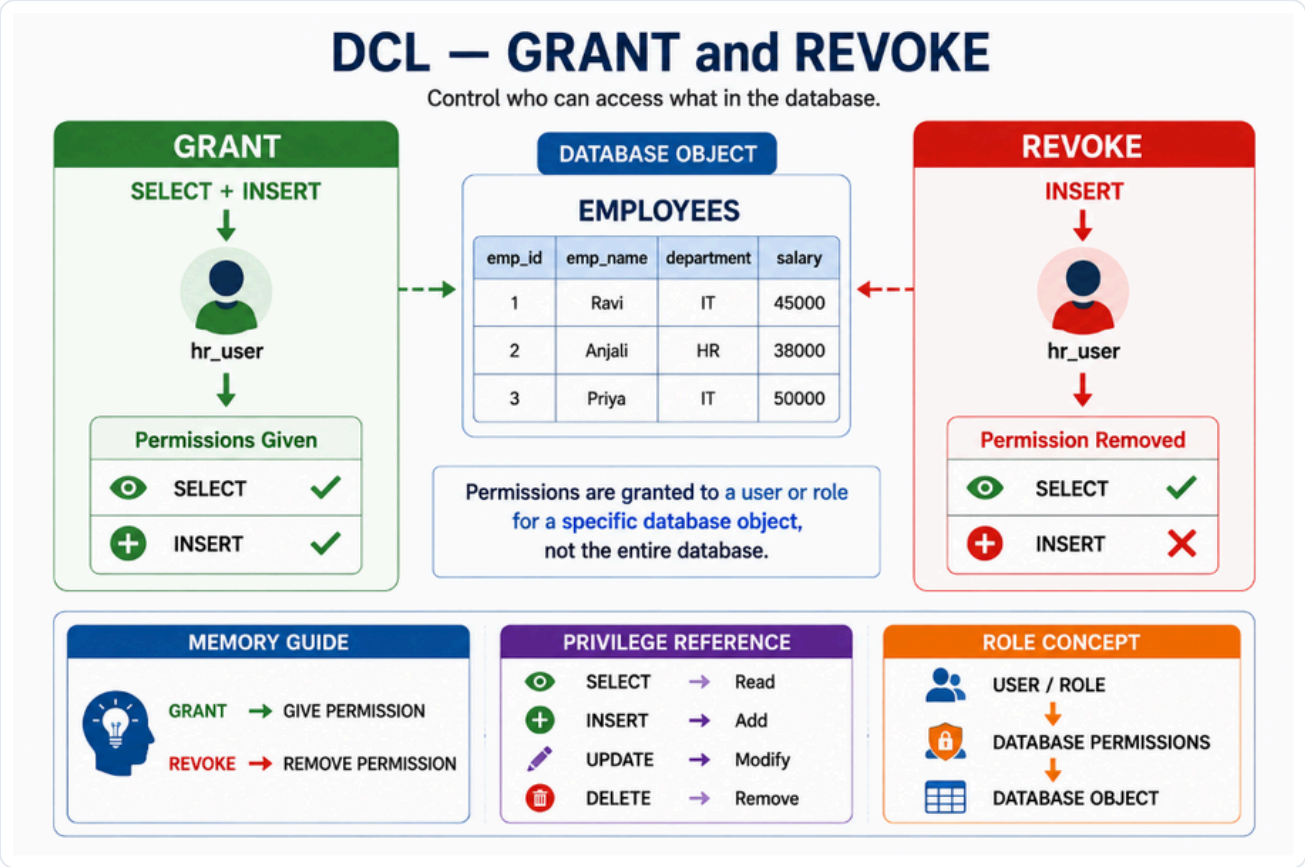


Figure 1: GRANT gives database permissions to a user or role, while REVOKE removes previously granted permissions.

GRANT

`GRANT` is used to **give a user or role permission** to perform specific operations on a database object.

Example:

```sql
GRANT SELECT, INSERT
ON employees
TO hr_user;
```

- Read data using `SELECT`
- Add data using `INSERT`

The exact syntax and available privileges can vary between database systems.

> *GRANT → Give permission*

REVOKE

`REVOKE` is used to **remove a previously granted permission**.

Example:

```SQL
REVOKE INSERT
ON employees
FROM hr_user;
```

After this, `hr_user` no longer has the `INSERT` permission from that grant, but may still have `SELECT` permission.

```TEXT
Before:
SELECT ✓
INSERT ✓

After REVOKE:
SELECT ✓
INSERT ✗
```

> *REVOKE → Remove permission*

Common Permissions

Database systems can provide different privileges depending on the product.

Common examples include:

```text
SELECT  → Read data
INSERT  → Add data
UPDATE  → Change data
DELETE  → Remove data
```

For example:

```sql
GRANT SELECT ON employees TO hr_user;
```

allows the role/user to read from `employees`, subject to the database's permission model.

Users and Roles

Permissions are commonly assigned to **users or roles**.

A role is a named collection of permissions that can be assigned to users.

For example:

```text
HR Role
   ↓
SELECT
INSERT
UPDATE
   ↓
HR Users
```

Using roles can make permission management easier when many users need similar access.

GRANT vs REVOKE

Command	Action	Example
`GRANT`	Give permission	`GRANT SELECT ...`
`REVOKE`	Remove permission	`REVOKE SELECT ...`

Easy Memory Trick

```text
GRANT  → GIVE
REVOKE → REMOVE
```

Why DCL Matters

Not every user should have unrestricted access to every database object.

For example:

```text
Student
→ View permitted student information

Teacher
→ View and update authorized academic data

Admin
→ Manage broader database access
```

Using permissions and roles helps follow the **principle of least privilege**:

> *Give users only the access they need to perform their job.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`GRANT` gives a user the entire database	It grants specified privileges on specified objects
`REVOKE` deletes the user	It removes a specified privilege
DCL changes table data	DCL manages access and permissions
`SELECT` means the user can modify data	`SELECT` normally provides read access
Every database uses exactly the same privilege syntax	Permission syntax and behavior vary by database product
Permissions should be given to everyone	Access should follow appropriate security requirements and least privilege

Placement Quick Points

```text
DCL
→ ACCESS CONTROL

GRANT
→ GIVE PERMISSION

REVOKE
→ REMOVE PERMISSION

SELECT
→ READ

INSERT
→ ADD

UPDATE
→ MODIFY

DELETE
→ REMOVE
```

Copy

- DCL is commonly used to manage database access and permissions.
- `GRANT` gives privileges to users or roles.
- `REVOKE` removes previously granted privileges.
- Permissions can be assigned to users or roles.
- Common privileges include `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Permission syntax and behavior can vary between database systems.
- Least privilege helps reduce unnecessary access.

Interview Questions

Q1. What is DCL?

> DCL stands for **Data Control Language** and is commonly used to manage database permissions and access.

Q2. What is GRANT?

> `GRANT` gives a user or role permission to perform specified operations on database objects.

Q3. What is REVOKE?

> `REVOKE` removes previously granted permissions.

Q4. What is the difference between GRANT and REVOKE?

```
text GRANT → Give permission REVOKE → Remove permission
```

Q5. What is a database role?

> A role is a named collection of permissions that can be assigned to users, making access management easier.

Q6. Why are database permissions important?

> They help prevent unauthorized users from viewing or modifying data and allow access to be controlled according to user responsibilities.

Practice & Hands-On Exercises

DCL commands require appropriate database users/roles and permissions, so they may not be executable in a shared learning environment.

1. Explain what this command does:

```
GRANT SELECT, INSERT
ON employees
TO hr_user;
```

1. Explain what this command does:

```
REVOKE INSERT
ON employees
FROM hr_user;
```

1. After the `REVOKE`, which permission does `hr_user` still have?
2. Explain why roles are useful when many users need the same permissions.
3. Give one example of a user who should have read-only access.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
DCL
 ↓
ACCESS CONTROL

GRANT
 ↓
GIVE PERMISSION

REVOKE
 ↓
REMOVE PERMISSION
```

DCL controls access to database objects: GRANT gives permissions, while REVOKE removes permissions.

End of DCL — GRANT, REVOKE

5.5 DML — INSERT, UPDATE, DELETE

Level: 5 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

DML (Data Manipulation Language) is used to **add, change, and remove data** stored in database tables.

The three main commands are:

Command	Purpose
`INSERT`	Add new rows
`UPDATE`	Change existing rows
`DELETE`	Remove rows

> ***DML works with the data inside a table, not the table structure itself.***

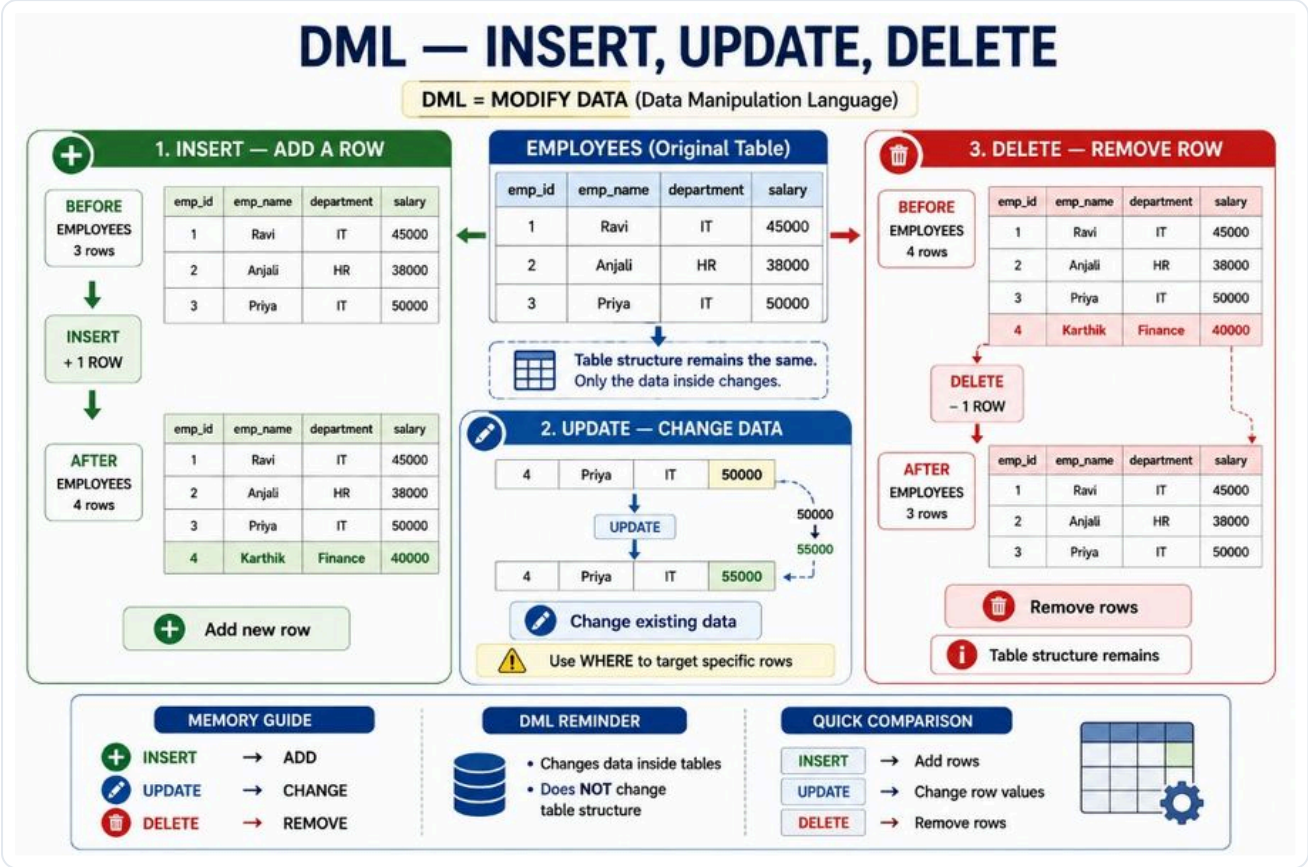


Figure 1: DML commands modify the data stored in a table—INSERT adds rows, UPDATE changes values, and DELETE removes rows.

Practice Table

This topic reuses the `employees` table from **5.2 DDL**.

emp_id	emp_name	department	salary
1	Ravi Kumar	IT	45000
2	Anjali Mehta	HR	38000
3	Suresh Rao	Finance	42000
4	Priya Nair	IT	50000

INSERT

`INSERT` is used to **add new rows** to a table.

```sql
INSERT INTO employees
(emp_id, emp_name, department, salary)
VALUES
(5, 'Karthik Iyer', 'Finance', 40000);
```

A new employee is added to the table.

> *INSERT → Add data*

UPDATE

`UPDATE` is used to **change existing data**.

```sql
UPDATE employees
SET salary = 55000
WHERE emp_id = 4;
```

Priya's salary changes from `50000` to `55000`.

Important

Be careful with the `WHERE` condition.

```sql
UPDATE employees
SET salary = 55000;
```

Without `WHERE`, this updates the salary for **every row**.

> *UPDATE → Change existing data*

DELETE

`DELETE` is used to **remove rows** from a table.

```sql
DELETE FROM employees
WHERE emp_id = 5;
```

This removes the employee whose ID is `5`.

Important

Without a `WHERE` condition:

```sql
DELETE FROM employees;
```

all rows are targeted for deletion. The table structure remains.

INSERT vs UPDATE vs DELETE

Command	What It Does	Table Structure
`INSERT`	Adds rows	Unchanged
`UPDATE`	Changes row values	Unchanged
`DELETE`	Removes rows	Unchanged

```text
INSERT → ADD
UPDATE → CHANGE
DELETE → REMOVE
```

DML and Transactions

DML operations can be part of a transaction.

For example, to discard an uncommitted change:

```sql
BEGIN;

UPDATE employees
SET salary = 55000
WHERE emp_id = 4;

ROLLBACK;
```

The uncommitted change is undone.

Or, to permanently apply it:

```sql
BEGIN;

UPDATE employees
SET salary = 55000
WHERE emp_id = 4;

COMMIT;
```

The change is committed.

> *DML changes can be controlled through transactions, depending on the database system and transaction settings.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`INSERT` changes table structure	`INSERT` adds rows
`UPDATE` adds a new row	`UPDATE` changes existing row values
`DELETE` removes the table	`DELETE` removes rows
`UPDATE` without `WHERE` changes one row	It can change all rows
`DELETE` without `WHERE` removes one row	It can remove all rows
DML and DDL are the same	DML changes stored data; DDL manages structure
`ROLLBACK` always works for every DML statement in exactly the same way	Transaction behavior can vary by database system and configuration

Placement Quick Points

```text
INSERT
→ ADD ROWS

UPDATE
→ CHANGE DATA

DELETE
→ REMOVE ROWS
```

- DML stands for **Data Manipulation Language**.

- `INSERT` adds new rows.
- `UPDATE` changes existing values.
- `DELETE` removes rows.
- `WHERE` is important when you want to target specific rows.
- DML operations can participate in transactions.

Interview Questions

Q1. What is DML?

DML stands for **Data Manipulation Language** and is used to add, modify, and remove data stored in tables.

Q2. What is INSERT used for?

`INSERT` adds new rows to a table.

Q3. What is UPDATE used for?

`UPDATE` changes values in existing rows.

Q4. What is DELETE used for?

`DELETE` removes rows from a table.

Q5. What happens if UPDATE is used without WHERE?

All rows that satisfy the statement's target are updated; for a simple table-wide update, that means every row.

Q6. What happens if DELETE is used without WHERE?

All rows in the targeted table are deleted.

Q7. Does DELETE remove the table?

> No. `DELETE` removes rows while the table structure remains.

Q8. What is the difference between DML and DDL?

> `text DML → Changes stored data DDL → Manages database structure`

Practice & Hands-On Exercises

Using the `employees` table:

1. Insert a new employee:

```
CODE
Copy

INSERT INTO employees (emp_id, emp_name, department, salary)
VALUES (5, 'Karthik Iyer', 'Finance', 40000);
```

1. Update an employee's salary:

```
CODE
Copy

UPDATE employees SET salary = 55000 WHERE emp_id = 4;
```

1. Update the department of one employee:

```
CODE
Copy

UPDATE employees SET department = 'Operations' WHERE emp_id = 2;
```

1. Delete one employee using `emp_id`:

```
CODE
Copy

DELETE FROM employees WHERE emp_id = 5;
```

1. Explain what happens if `WHERE` is removed from an `UPDATE`.

2. Explain what happens if `WHERE` is removed from a `DELETE`.
3. Write one `INSERT`, one `UPDATE`, and one `DELETE` statement.
4. Explain the difference between DML and DDL.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE Copy

```
INSERT → ADD
UPDATE → CHANGE
DELETE → REMOVE
```

> ***DML commands modify the data inside tables: INSERT adds rows, UPDATE changes existing data, and DELETE removes rows.***

End of DML — INSERT, UPDATE, DELETE

5.6 TCL — COMMIT, ROLLBACK, SAVEPOINT

Level: 5 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

TCL (Transaction Control Language) is commonly used to control database transactions.

The main commands are:

Command	Purpose
`COMMIT`	Save transaction changes
`ROLLBACK`	Undo uncommitted changes
`SAVEPOINT`	Create a point to roll back to

Think of a transaction like a series of steps:

```text
BEGIN
  ↓
SQL OPERATIONS
  ↓
COMMIT / ROLLBACK
```

> *TCL controls what happens to changes made during a transaction.*

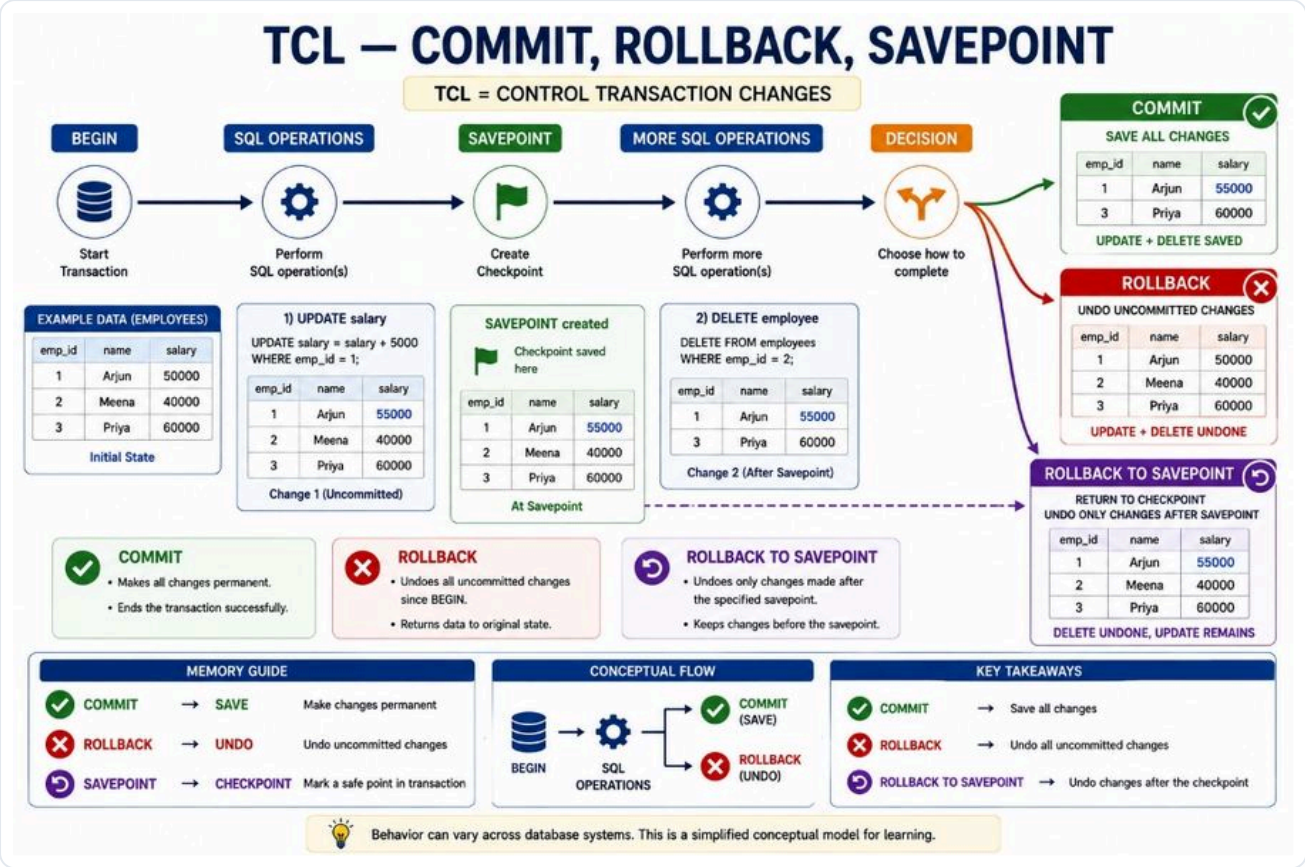


Figure 1: TCL commands control transaction changes by saving them, undoing them, or returning to a specific checkpoint.

COMMIT

`COMMIT` permanently saves the changes made during the current transaction.

Example:

After `COMMIT`, the salary change becomes part of the committed database state.

> *COMMIT → Save changes*

ROLLBACK

`ROLLBACK` undoes uncommitted changes made during the current transaction.

Example:

```sql
BEGIN;

DELETE FROM employees
WHERE emp_id = 2;

ROLLBACK;
```

The `DELETE` is undone, so the row for employee `2` remains.

> *ROLLBACK → Undo uncommitted changes*

SAVEPOINT

A `SAVEPOINT` creates a **checkpoint inside a transaction**.

You can roll back to that point without undoing all earlier changes in the transaction.

```sql
BEGIN;

UPDATE employees
SET salary = 46000.00
WHERE emp_id = 3;

SAVEPOINT after_salary_update;

DELETE FROM employees
WHERE emp_id = 3;

ROLLBACK TO after_salary_update;

COMMIT;
```

Here:

- The salary update remains.
- The delete is undone.
- The final `COMMIT` saves the remaining change.

Simple Flow

```text
UPDATE
  ↓
SAVEPOINT
  ↓
DELETE
  ↓
ROLLBACK TO SAVEPOINT
  ↓
DELETE UNDONE
  ↓
COMMIT
```

> *SAVEPOINT → Create a checkpoint inside a transaction*

COMMIT vs ROLLBACK vs SAVEPOINT

Command	Main Action
`COMMIT`	Save transaction changes
`ROLLBACK`	Undo uncommitted changes
`SAVEPOINT`	Create a rollback checkpoint

Easy Memory Trick

```text
COMMIT     → SAVE
ROLLBACK   → UNDO
SAVEPOINT  → CHECKPOINT
```

Important Difference

Consider:

```text
BEGIN
  ↓
UPDATE
  ↓
SAVEPOINT
  ↓
DELETE
  ↓
ROLLBACK TO SAVEPOINT
```

Only the work **after the savepoint** is undone.

But:

```
TEXT

BEGIN
  ↓
UPDATE
  ↓
DELETE
  ↓
ROLLBACK
```

Copy

rolls back the entire uncommitted transaction.

So:

> ***ROLLBACK → Undo the transaction's uncommitted changes***
> ***ROLLBACK TO SAVEPOINT → Undo changes back to a specific checkpoint***

Practice Table

This topic reuses the `employees` table from **5.2 DDL**.

emp_id	emp_name	department	salary
1	Ravi Kumar	IT	45000
2	Anjali Mehta	HR	38000
3	Suresh Rao	Finance	42000
4	Priya Nair	IT	50000

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`COMMIT` undoes changes	`COMMIT` saves the transaction
`ROLLBACK` saves changes	`ROLLBACK` undoes uncommitted changes
`SAVEPOINT` commits a transaction	It only creates a checkpoint inside the transaction
`ROLLBACK TO` undoes the entire transaction	It rolls back to the specified savepoint
`COMMIT` can normally be followed by `ROLLBACK` to undo it	Normal rollback cannot undo already committed work
TCL changes table structure	TCL controls transaction changes

Placement Quick Points

```text
COMMIT
→ SAVE CHANGES

ROLLBACK
→ UNDO UNCOMMITTED CHANGES

SAVEPOINT
→ CREATE CHECKPOINT
```

- TCL is commonly used to control database transactions.
- `COMMIT` saves transaction changes.

- `ROLLBACK` undoes uncommitted transaction changes.
- `SAVEPOINT` creates a checkpoint within a transaction.
- `ROLLBACK TO SAVEPOINT` returns the transaction to that checkpoint.
- Transaction behavior can vary depending on the database system and transaction settings.

Interview Questions

Q1. What is TCL?

TCL stands for **Transaction Control Language** and is commonly used to control transaction changes.

Q2. What does COMMIT do?

`COMMIT` saves the changes made by the current transaction.

Q3. What does ROLLBACK do?

`ROLLBACK` undoes uncommitted changes made during the current transaction.

Q4. What is a SAVEPOINT?

A savepoint is a checkpoint created inside a transaction so that the transaction can be rolled back to that point.

Q5. What is the difference between ROLLBACK and ROLLBACK TO SAVEPOINT?

```text ROLLBACK → Undo uncommitted transaction changes

ROLLBACK TO SAVEPOINT → Undo changes after a specific checkpoint ```
```

Q6. What happens after COMMIT?

> The transaction's changes are committed and become part of the database's committed state.

Q7. Give a real-world example of SAVEPOINT.

> A long transaction may contain several steps. A savepoint allows the application to undo only the later steps while keeping earlier work.

Practice & Hands-On Exercises

Using the `employees` table:

1. Update an employee's salary and use `COMMIT`:

```
CODE
Copy

BEGIN;
UPDATE employees SET salary = 48000.00 WHERE emp_id = 1;
COMMIT;
```

1. Delete an employee and use `ROLLBACK`:

```
CODE
Copy

BEGIN;
DELETE FROM employees WHERE emp_id = 2;
ROLLBACK;
```

1. Perform an update, create a savepoint, perform another change, and use `ROLLBACK TO` the savepoint:

```
BEGIN;
UPDATE employees SET salary = 46000.00 WHERE emp_id = 3;
SAVEPOINT after_salary_update;
DELETE FROM employees WHERE emp_id = 3;
ROLLBACK TO after_salary_update;
COMMIT;
```

1. Explain which changes remain after the final `COMMIT`.
2. Explain the difference between `ROLLBACK` and `ROLLBACK TO SAVEPOINT`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
COMMIT
→ SAVE

ROLLBACK
→ UNDO

SAVEPOINT
→ CHECKPOINT
```

> ***TCL controls transaction changes: COMMIT saves them, ROLLBACK undoes uncommitted changes, and SAVEPOINT creates a checkpoint for partial rollback.***

End of TCL — COMMIT, ROLLBACK, SAVEPOINT

6.1 Constraints (NOT NULL, UNIQUE, PK, FK, CHECK, DEFAULT)

Level: 6 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

SQL **constraints** are rules applied to table columns to help keep data **valid, consistent, and reliable**.

The six common constraints covered here are:

Constraint	Purpose
`NOT NULL`	Value must be provided
`UNIQUE`	Prevents duplicate non-NULL values in a column/column set
`PRIMARY KEY`	Uniquely identifies each row
`FOREIGN KEY`	Maintains relationships between tables
`CHECK`	Requires values to satisfy a condition
`DEFAULT`	Supplies a value when one is not provided

> *Constraints help the database prevent invalid data from being stored.*

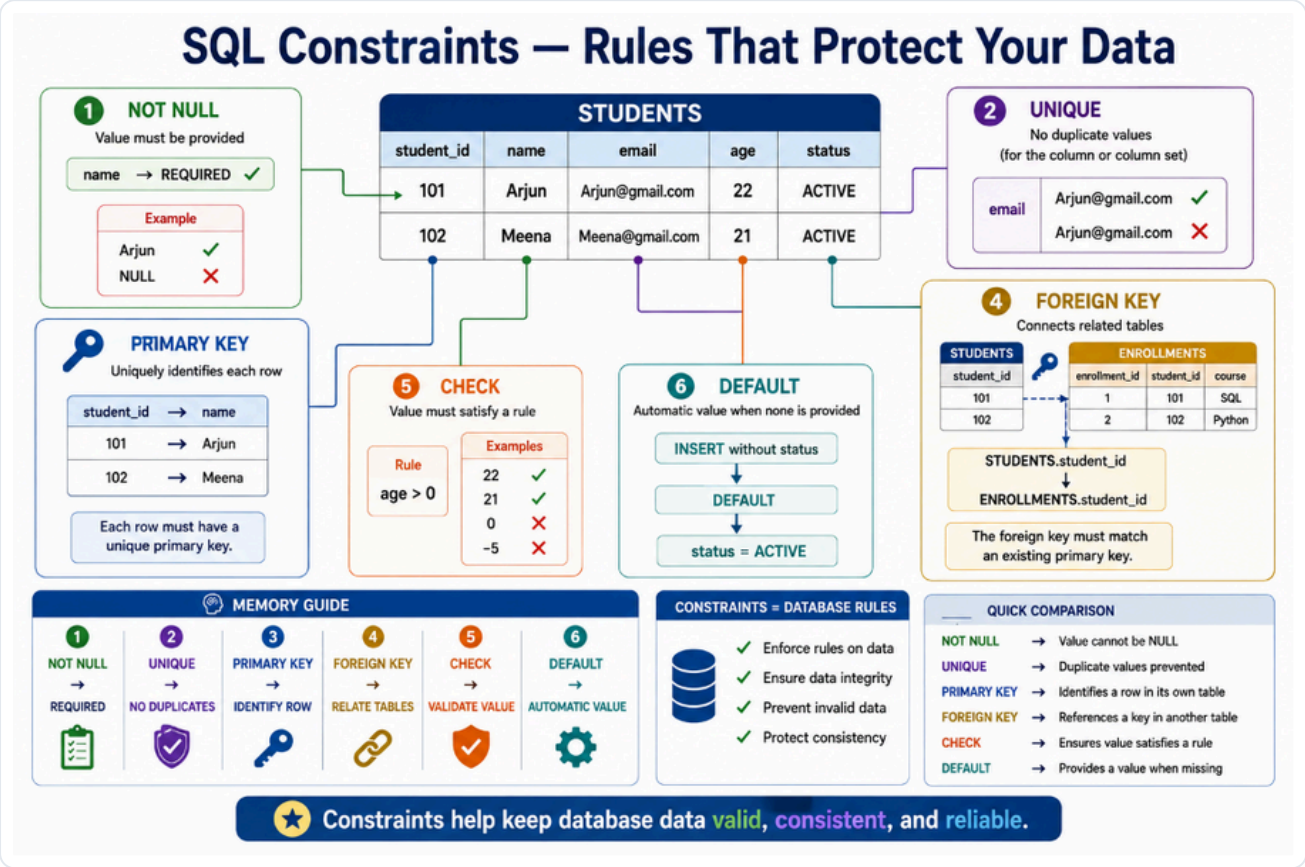


Figure 1: SQL constraints enforce rules that help maintain valid, consistent, and reliable table data.

NOT NULL

`NOT NULL` ensures that a column cannot contain a `NULL` value.

Example:

```sql
name VARCHAR(100) NOT NULL
```

This means a student must have a value in the `name` column.

> *NOT NULL → Value is required*

UNIQUE

`UNIQUE` prevents duplicate values for the constrained column or column combination.

Example:

```SQL
email VARCHAR(150) UNIQUE
```

Two students should not have the same email address under this constraint.

> *UNIQUE → Prevent duplicate values*

Note: `NULL` handling for `UNIQUE` constraints can vary depending on the database system.

PRIMARY KEY

A **PRIMARY KEY** uniquely identifies each row in a table.

Example:

```SQL
student_id INT PRIMARY KEY
```

```text
student_id
101 → Arjun
102 → Meena
103 → Ravi
```

> *PRIMARY KEY → Unique identity of a row*

A table has one primary key constraint, which can consist of one column or multiple columns.

FOREIGN KEY

A **FOREIGN KEY** creates a relationship between tables by referencing a key in another table.

Example:

```sql
CREATE TABLE enrollments (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_name VARCHAR(100),
    FOREIGN KEY (student_id)
        REFERENCES students(student_id)
);
```

Here:

```text
students.student_id
        ↑
        |
enrollments.student_id
```

The foreign key helps prevent a child row from referencing a parent key that does not exist, subject to the configured foreign-key actions.

> *FOREIGN KEY → Connect related tables*

CHECK

`CHECK` requires a value to satisfy a specified condition.

Example:

```sql
age INT CHECK (age > 0)
```

This prevents values that violate the condition.

For example:

```text
Age = 22  ✓
Age = 0   X
Age = -5  X
```

> *CHECK → Value must satisfy a rule*

DEFAULT

Example:

```sql
status VARCHAR(20) DEFAULT 'ACTIVE'
```

Then:

```sql
INSERT INTO students (name, email, age)
VALUES ('Arjun', 'arjun@gmail.com', 22);
```

The database can automatically use:

```text
status = ACTIVE
```

> *DEFAULT → Use a value automatically when none is supplied*

Practice Table Setup

This table demonstrates:

```
TEXT

student_id → PRIMARY KEY
name       → NOT NULL
email      → UNIQUE
age        → CHECK
status     → DEFAULT
```

FOREIGN KEY Example

```sql
SQL

CREATE TABLE enrollments (
    enrollment_id SERIAL PRIMARY KEY,
    student_id INT REFERENCES students(student_id),
    course_name VARCHAR(100)
);
```

Here `enrollments.student_id` references `students.student_id`.

A student ID that does not exist in `students` cannot normally be inserted into `enrollments`.

Constraint Comparison

Constraint	Main Question It Answers
`NOT NULL`	Must a value be provided?
`UNIQUE`	Can this value be duplicated?
`PRIMARY KEY`	Which value uniquely identifies this row?
`FOREIGN KEY`	Does this value refer to a valid related row?
`CHECK`	Does this value satisfy the rule?
`DEFAULT`	What value should be used when none is provided?

Easy Memory Trick

```text
NOT NULL    → REQUIRED
UNIQUE      → NO DUPLICATES
PRIMARY KEY → IDENTIFY
FOREIGN KEY → RELATE
CHECK       → RULE
DEFAULT     → AUTOMATIC VALUE
```

Constraints with ALTER TABLE

Constraints can also be added to an existing table using `ALTER TABLE`.

For example:

```sql
ALTER TABLE students
ADD CONSTRAINT chk_age
CHECK (age > 0);
```

The exact syntax can vary between database systems.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`NOT NULL` means the value must be unique	It only prevents `NULL` values
`UNIQUE` identifies the row exactly like a primary key	A primary key is the table's designated unique row identifier
A foreign key must always reference a primary key	It references a candidate/unique key according to the database's rules
`CHECK` automatically fixes invalid values	It rejects values that violate its condition
`DEFAULT` replaces every missing value with the default	It is used when a value is not supplied for the column
Constraints are only checked when a table is created	They are enforced when relevant data changes are attempted
`PRIMARY KEY` can contain `NULL`	A primary key cannot contain `NULL`

Placement Quick Points

```text
NOT NULL
→ REQUIRED VALUE

UNIQUE
→ NO DUPLICATE VALUES

PRIMARY KEY
→ UNIQUE ROW IDENTIFIER

FOREIGN KEY
→ TABLE RELATIONSHIP

CHECK
→ VALIDATION RULE

DEFAULT
→ AUTOMATIC VALUE
```

Copy

- Constraints help maintain data integrity.
- `NOT NULL` prevents `NULL` values.
- `UNIQUE` enforces uniqueness for the constrained key.
- `PRIMARY KEY` uniquely identifies rows.
- `FOREIGN KEY` maintains relationships between tables.
- `CHECK` enforces a condition.
- `DEFAULT` supplies a value when one is not provided.

Interview Questions

Q1. What is a SQL constraint?

> A constraint is a rule applied to database data to help maintain validity and integrity.

Q2. What is the difference between `UNIQUE` and `PRIMARY KEY`?

> Both enforce uniqueness, but a primary key is the table's designated row identifier and cannot contain `NULL`. A table has one primary key constraint, while it can have multiple unique constraints.

Q3. What is a FOREIGN KEY?

> A foreign key is a column or set of columns that references a key in another table to enforce a relationship.

Q4. What is the purpose of CHECK?

> `CHECK` ensures that values satisfy a specified condition.

Q5. What does DEFAULT do?

> `DEFAULT` supplies a value when an `INSERT` does not provide one for that column.

Q6. Can a PRIMARY KEY contain NULL?

> No.

Q7. Why are constraints important?

> They allow the database itself to enforce important data rules instead of relying only on application code.

Q8. Can constraints be added after table creation?

> Yes. Many database systems allow constraints to be added or modified using `ALTER TABLE`, although exact syntax varies.

Practice & Hands-On Exercises

Using the `students` and `enrollments` tables:

1. Identify each constraint in the `students` table.

2. Try inserting a student without a `name`.
3. Try inserting a duplicate email:

```sql
INSERT INTO students (name, email, age)
VALUES ('Duplicate User', 'arjun@gmail.com', 25);
```

1. Try inserting an invalid age:

```sql
INSERT INTO students (name, email, age)
VALUES ('Invalid Age', 'invalid@gmail.com', -5);
```

1. Insert a student without `status` and observe the default value:

```sql
INSERT INTO students (name, email, age)
VALUES ('New Student', 'new@gmail.com', 20);
SELECT * FROM students WHERE email = 'new@gmail.com';
```

1. Try inserting an enrollment with a non-existing `student_id`:

```sql
INSERT INTO enrollments (student_id, course_name)
VALUES (999, 'Machine Learning');
```

1. Explain the difference between `PRIMARY KEY` and `FOREIGN KEY`.
2. Explain the difference between `UNIQUE` and `NOT NULL`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```TEXT
NOT NULL     → REQUIRED
UNIQUE       → NO DUPLICATES
PRIMARY KEY  → IDENTIFY
FOREIGN KEY  → RELATE
CHECK        → VALIDATE
DEFAULT      → FILL AUTOMATICALLY
```

> ***SQL constraints are database rules that help keep data valid, unique, consistent, and correctly related.***

End of Constraints (NOT NULL, UNIQUE, PK, FK, CHECK, DEFAULT)

6.2 Database Objects (Table, View, Materialized View, Sequence, Function)

Level: 6 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **database object** is a defined object created and managed inside a database.

In this topic, we will learn five useful database objects:

Object	Main Purpose
Table	Stores data
View	Provides a virtual result from a query
Materialized View	Stores a query result for reuse
Sequence	Generates numeric values
Function	Reusable database logic that returns a value

A simple way to remember them:

```text
TABLE             → STORE
VIEW              → SHOW
MATERIALIZED VIEW → STORE RESULT
SEQUENCE          → GENERATE
FUNCTION          → REUSE LOGIC
```

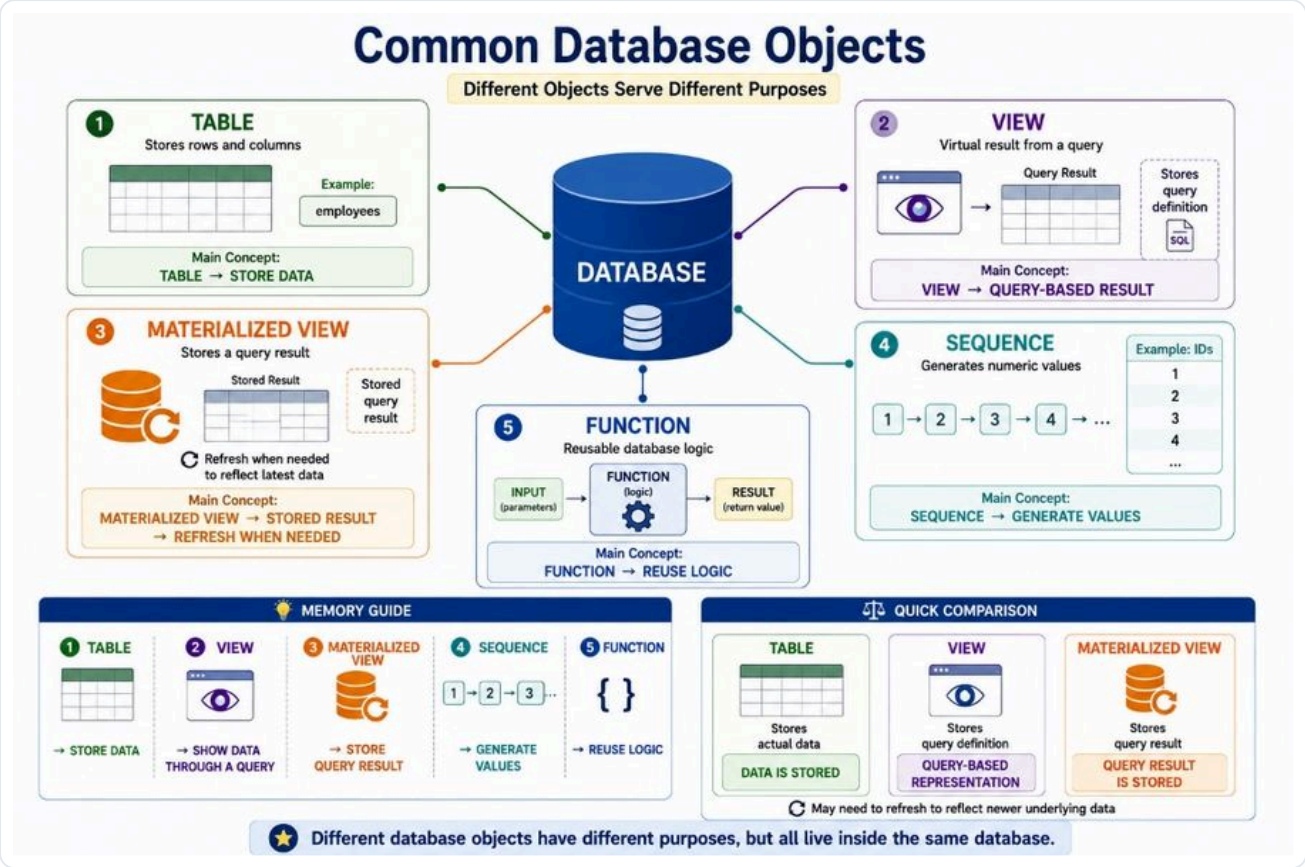


Figure 1: Database objects serve different purposes: tables store data, views present query-based results, materialized views store query results, sequences generate values, and functions provide reusable database logic.

1. Table

A **table** is the main structure used to store relational data in rows and columns.

Example:

Example data:

student_id	name	age	email
101	Arun	20	arun@gmail.com
102	Meena	21	meena@gmail.com

> *Table → Stores actual data*

---

2. View

A **view** is a named query that presents data from one or more tables.

It is often described as a **virtual table** because the view definition stores the query rather than a separate copy of the underlying result.

For example:

```sql
CREATE VIEW employee_department_view AS
SELECT e.emp_id, e.emp_name, d.dept_name, s.salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
JOIN salaries s ON e.emp_id = s.emp_id;
```

You can then query it like a table:

```sql
SELECT * FROM employee_department_view;
```

The underlying query is evaluated according to the database system when the view is queried.

> *View → Reusable query-based representation*

---

3. Materialized View

A **materialized view** stores the result of a query so it can be read without recomputing the full underlying query every time.

For example:

```SQL
CREATE MATERIALIZED VIEW mv_employee_details AS
SELECT e.emp_id, e.emp_name, d.dept_name, s.salary
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
JOIN salaries s ON e.emp_id = s.emp_id;
```

The materialized result must be refreshed to reflect newer underlying data:

```SQL
REFRESH MATERIALIZED VIEW mv_employee_details;
```

> *Materialized View → Store query result for faster/reusable reads*

View vs Materialized View

Feature	View	Materialized View
What is Stored	Stores the query definition	Stores the query result
Computation	Result is generally computed when queried	Result is precomputed and stored on disk
Freshness	Reflects current underlying data immediately	May become stale until explicitly refreshed
Storage Usage	Uses minimal storage (definition only)	Requires physical storage for the materialized data

4. Sequence

A **sequence** is a database object that generates a sequence of numeric values.

It is commonly used when applications need generated identifiers.

For example:

```sql
CREATE SEQUENCE emp_seq
START WITH 1
INCREMENT BY 1;
```

Values can then be obtained using database-specific sequence syntax. In PostgreSQL:

```sql
SELECT nextval('emp_seq');
```

Possible results:

```
1
2
3
4
```

> *Sequence → Generate numeric values*

Important: Sequence behavior, syntax, and use with transactions vary between database systems.

---

5. Function

A **function** is a reusable block of database logic that can accept inputs and return a value or result.

For example, in PostgreSQL:

The function can be called like:

```sql
SELECT get_employee_salary(2);
```

It returns the salary associated with employee 2.

> ***Function → Reusable database logic***

Note: Whether and how functions can perform INSERT, UPDATE, or DELETE depends on the database system and function type. Do not treat "functions cannot perform DML" as a universal SQL rule.

Practice Tables

We will use `departments`, `employees`, and `salaries`:

```sql
CREATE TABLE departments (
    dept_id SERIAL PRIMARY KEY,
    dept_name VARCHAR(50)
);

INSERT INTO departments (dept_name) VALUES
('HR'), ('IT'), ('Finance');

CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT REFERENCES departments(dept_id)
);

INSERT INTO employees (emp_name, dept_id) VALUES
('Ravi', 2),
('Anita', 2),
('Kiran', 2),
('Meena', 3);

CREATE TABLE salaries (
    emp_id INT REFERENCES employees(emp_id),
    salary NUMERIC(10, 2)
);

INSERT INTO salaries (emp_id, salary) VALUES
(1, 45000.00),
(2, 60000.00),
(3, 55000.00),
(4, 50000.00);
```

Table vs View vs Materialized View

These three are commonly confused:

```text
TABLE             → STORES DATA
VIEW              → STORES QUERY DEFINITION
MATERIALIZED VIEW → STORES QUERY RESULT
```

This is the most important distinction to remember.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
A view stores a separate copy of all result data	A normal view generally stores its query definition
A materialized view always contains the latest data	It may need to be refreshed
A sequence is a table	It is a separate object used to generate values
A function is the same as a view	A function contains reusable logic and can accept parameters/return values
All database systems support these objects in exactly the same way	Syntax and behavior vary between database products
A materialized view automatically updates whenever source tables change	Refresh behavior depends on the database system and configuration

Placement Quick Points

```text
TABLE
→ STORE DATA

VIEW
→ QUERY-BASED VIRTUAL RESULT

MATERIALIZED VIEW
→ STORED QUERY RESULT

SEQUENCE
→ GENERATE NUMERIC VALUES

FUNCTION
→ REUSABLE DATABASE LOGIC
```

Copy

- A **table** stores relational data.
- A **view** provides a query-based representation of data.
- A **materialized view** stores a query result and may need refreshing.
- A **sequence** generates numeric values.
- A **function** contains reusable database logic and can return a value/result.
- Exact syntax and behavior vary across database systems.

Interview Questions

Q1. What is a database object?

> A database object is a defined object managed by a database system, such as a table, view, sequence, or function.

Q2. What is the difference between a table and a view?

> A table stores data directly, while a view provides a query-based representation of data without storing rows separately.

Q3. What is the difference between a view and a materialized view?

> A normal view stores the query definition, while a materialized view stores the query result and can be refreshed.

Q4. What is a sequence?

> A sequence is a database object used to generate a series of numeric values.

Q5. Why are sequences useful?

> They are useful when applications need generated numeric values, such as primary key identifiers.

Q6. What is a database function?

> A function is reusable database logic that can accept parameters and return a value or result.

Q7. Which database object is commonly used to store relational data?

> A table.

Q8. Which object can store a precomputed query result?

> A materialized view.

Practice & Hands-On Exercises

Using the examples above:

1. Identify the purpose of a table, view, materialized view, sequence, and function.
2. Explain the difference between a view and a materialized view.

3. Create a simple view showing employee names and departments:

```sql
CREATE VIEW emp_dept_simple AS
SELECT e.emp_name, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id;

SELECT * FROM emp_dept_simple;
```

1. Explain why a materialized view may need refreshing.
2. Create a sequence that starts at 100 and increases by 1:

```sql
CREATE SEQUENCE test_seq START WITH 100 INCREMENT BY 1;
SELECT nextval('test_seq');
```

1. Explain one real-world use of a database function.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
TABLE             → STORE DATA
VIEW              → SHOW DATA THROUGH A QUERY
MATERIALIZED VIEW → STORE QUERY RESULT
SEQUENCE          → GENERATE VALUES
FUNCTION          → REUSE LOGIC
```

> ***Database objects serve different purposes: tables store data, views present data, materialized views store query results, sequences generate values, and functions provide reusable database logic.***

End of Database Objects (Table, View, Materialized View, Sequence, Function)

6.3 Aggregate Functions (MIN, MAX, SUM, AVG, COUNT)

Level: 6 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

An **aggregate function** performs a calculation on multiple rows and returns a **single result**.

The five core aggregate functions are:

Function	Purpose
`MIN()`	Finds the smallest value
`MAX()`	Finds the largest value
`SUM()`	Calculates the total
`AVG()`	Calculates the average
`COUNT()`	Counts rows or non-NULL values

> *Aggregate functions summarize data and return a single value.*

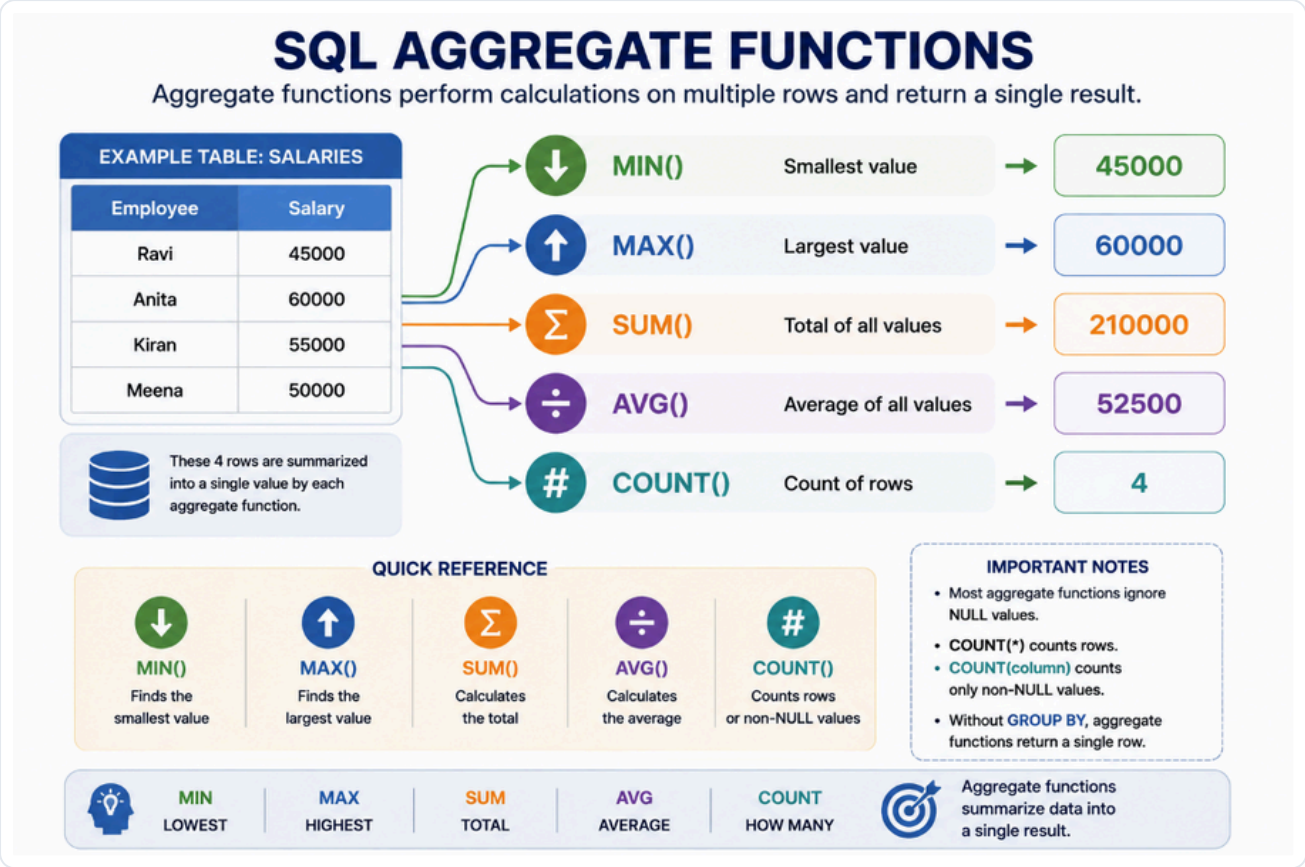


Figure 1: Aggregate functions summarize multiple rows and return a single result.

Practice Tables

This topic reuses the `employees` and `salaries` tables from **6.2 Database Objects**.

Salaries

emp_id	salary
1	45000
2	60000
3	55000
4	50000

MIN() and MAX()

`MIN()` returns the smallest value, while `MAX()` returns the largest value.

```sql
SELECT
    MIN(salary) AS minimum_salary,
    MAX(salary) AS maximum_salary
FROM salaries;
```

Result:

minimum_salary	maximum_salary
45000	60000

> *MIN() → Smallest value*
> *MAX() → Largest value*

SUM()

`SUM()` calculates the total of the selected numeric values.

```sql
SELECT SUM(salary) AS total_salary
FROM salaries;
```

Result:

total_salary
210000

> ***SUM() → Total***

AVG()

`AVG()` calculates the average of the selected numeric values.

```sql
SELECT AVG(salary) AS average_salary
FROM salaries;
```

Result:

average_salary
52500

> *AVG() → Average*

COUNT()

`COUNT()` counts rows or non-NULL values, depending on how it is used.

Count all rows

```sql
SELECT COUNT(*) AS employee_count
FROM employees;
```

Result:

employee_count
4

Count non-NULL salary values

```sql
SELECT COUNT(salary) AS salary_count
FROM salaries;
```

`COUNT(salary)` counts only rows where `salary` is not `NULL`.

> *COUNT(*) → Counts rows*
> *COUNT(column) → Counts non-NULL values*

NULL and Aggregate Functions

A common beginner point is that most aggregate functions ignore `NULL` values.

For example, consider four salary rows:

```text
Salary:
45000
60000
NULL
50000
```

`SUM(salary)` does not treat `NULL` as zero; the `NULL` value is ignored in the aggregate calculation.

Similarly:

```text
COUNT(salary) → 3
COUNT(*)      → 4
```

> *`COUNT(*)` counts rows, while `COUNT(column)` counts non-NULL values in that column.*

Simple Comparison

Function	Example Question
`MIN()`	What is the lowest salary?
`MAX()`	What is the highest salary?
`SUM()`	What is the total salary?
`AVG()`	What is the average salary?
`COUNT()`	How many employees are there?

Easy Memory Trick

```text
MIN   → LOWEST
MAX   → HIGHEST
SUM   → TOTAL
AVG   → AVERAGE
COUNT → HOW MANY
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`COUNT(*)` and `COUNT(column)` always return the same result	`COUNT(column)` ignores `NULL`; `COUNT(*)` counts rows
`AVG()` treats `NULL` as zero	`NULL` values are ignored by `AVG()`
`SUM()` works on text values	It is normally used with numeric values
Aggregate functions always return multiple rows	Without grouping, they normally return one result row
`MIN()` and `MAX()` only work with numbers	They can also be used with other comparable data types, depending on the database
`COUNT()` always counts values	Its behavior depends on whether `*`, a column, or an expression is used

Placement Quick Points

```text
MIN()   → SMALLEST
MAX()   → LARGEST
SUM()   → TOTAL
AVG()   → AVERAGE
COUNT() → COUNT
```

- Aggregate functions summarize multiple rows.
- `MIN()` finds the smallest value.

- `MAX()` finds the largest value.
- `SUM()` calculates a total.
- `AVG()` calculates an average.
- `COUNT(*)` counts rows.
- `COUNT(column)` counts non-NULL values.

Interview Questions

Q1. What is an aggregate function?

An aggregate function performs a calculation on multiple rows and returns a summarized result.

Q2. What are the five common aggregate functions?

```
text MIN() MAX() SUM() AVG() COUNT()
```

Q3. What is the difference between COUNT(*) and COUNT(column)?

`COUNT(*)` counts rows, while `COUNT(column)` counts only non-NULL values in that column.

Q4. What does AVG() do with NULL values?

`AVG()` ignores `NULL` values when calculating the average.

Q5. Which function finds the highest salary?

`MAX(salary)`.

Q6. Which function calculates total salary?

`SUM(salary)`.

Q7. Can aggregate functions be used with GROUP BY?

Yes. `GROUP BY` allows aggregate calculations to be performed separately for each group.

Example:

`Sql SELECT department, AVG(salary) FROM employees GROUP BY department;`

Practice & Hands-On Exercises

Using the `salaries` and `employees` tables:

1. Find the minimum salary:

```
CODE
SELECT MIN(salary) FROM salaries;
```

1. Find the maximum salary:

```
CODE
SELECT MAX(salary) FROM salaries;
```

1. Calculate the total salary:

```
CODE
SELECT SUM(salary) FROM salaries;
```

1. Calculate the average salary:

```
CODE
SELECT AVG(salary) FROM salaries;
```

1. Count all employees:

```
CODE
Copy

SELECT COUNT(*) FROM employees;
```

1. Explain the difference between `COUNT(*)` and `COUNT(salary)`.
2. Add a `NULL` salary and observe how `AVG()` behaves.
3. Calculate the average salary for each department using `GROUP BY`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE
Copy

MIN   → LOWEST
MAX   → HIGHEST
SUM   → TOTAL
AVG   → AVERAGE
COUNT → HOW MANY
```

> ***Aggregate functions summarize data by calculating values such as minimum, maximum, total, average, and count.***

End of Aggregate Functions (MIN, MAX, SUM, AVG, COUNT)

6.4 Stored Procedures

Level: 6 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **stored procedure** is a reusable set of SQL statements stored in the database and executed using a `CALL` statement.

It can accept parameters and perform multiple operations as one reusable unit.

```text
INPUT PARAMETERS
      ↓
STORED PROCEDURE
      ↓
SQL OPERATIONS
      ↓
RESULT / DATA CHANGES
```

> *Stored Procedure → Reusable database logic*

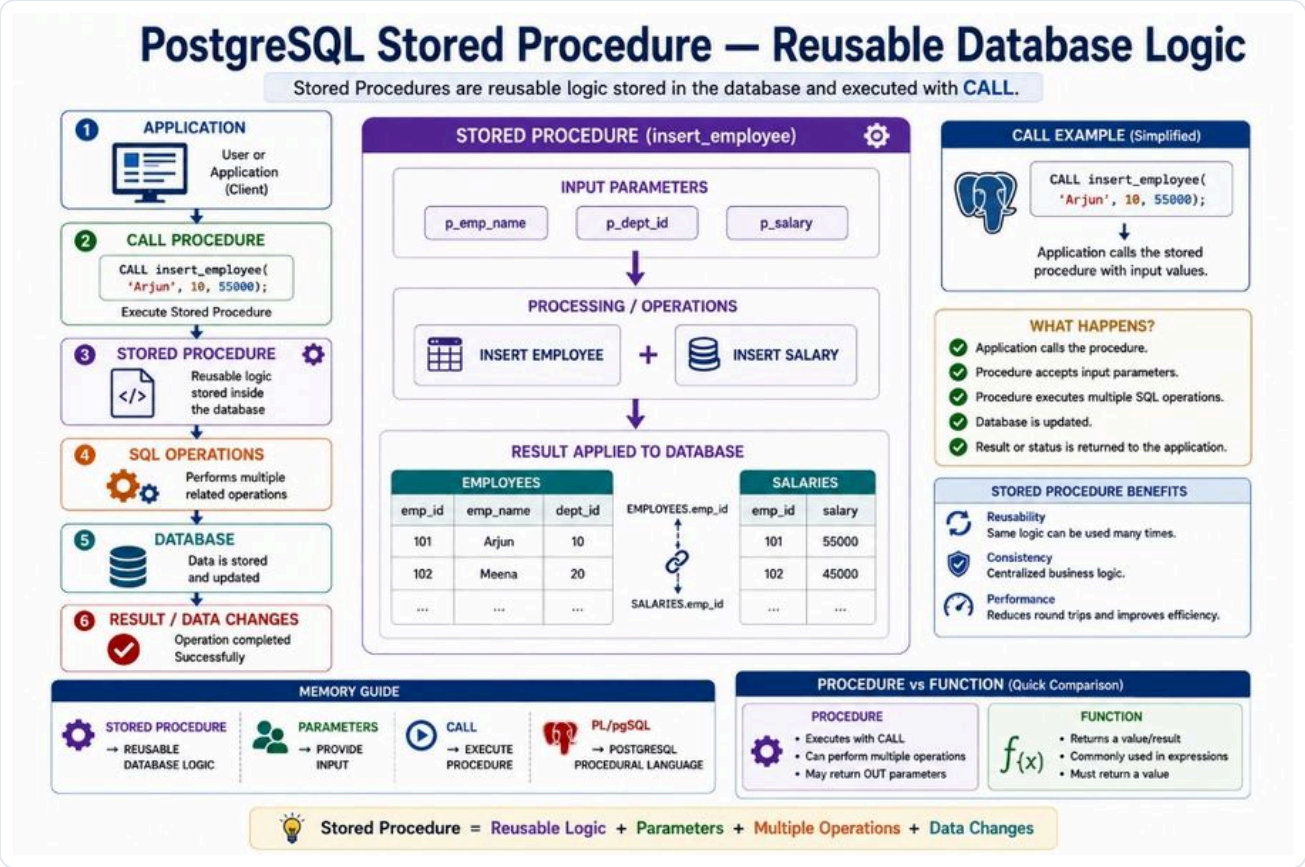


Figure 1: A PostgreSQL stored procedure accepts parameters and executes reusable database logic, including multiple SQL operations, through a single CALL.

PostgreSQL Example

Create a procedure that inserts an employee and their salary:

```sql
CREATE OR REPLACE PROCEDURE insert_employee(
    p_emp_name VARCHAR,
    p_dept_id INT,
    p_salary NUMERIC
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_emp_id INT;
BEGIN
    INSERT INTO employees (emp_name, dept_id)
    VALUES (p_emp_name, p_dept_id)
    RETURNING emp_id INTO v_emp_id;

    INSERT INTO salaries (emp_id, salary)
    VALUES (v_emp_id, p_salary);
END;
$$;
```

Here, the procedure accepts:

```text
p_emp_name → Employee name
p_dept_id  → Department ID
p_salary   → Salary
```

It then performs two related inserts.

Calling the Procedure

Use `CALL` to execute the procedure:

```sql
CALL insert_employee('Rahul', 2, 65000);
```

This executes the stored logic and adds the employee and salary.

> *CALL → Execute the stored procedure*

Why Use Stored Procedures?

Stored procedures are useful when the same database logic needs to be executed repeatedly.

They can:

- Accept parameters
- Perform multiple SQL operations
- Reuse business logic
- Reduce repeated SQL in applications
- Centralize database-side operations

Procedure vs Function

A procedure and function are related but not identical.

Feature	Procedure	Function
Invocation	Executed with `CALL`	Commonly used in expressions such as `SELECT`
Operations	Can perform procedural/database operations	Returns a value or result
Best Used For	Multi-step operations and workflows	Reusable calculations, formatting, or scalar logic

Note: Exact capabilities vary by database system.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
A procedure is the same as a table	A procedure stores reusable database logic
A procedure is called with `SELECT`	PostgreSQL procedures are invoked with `CALL`
A procedure can never accept parameters	Procedures can accept parameters
A procedure must contain only one SQL statement	It can contain multiple statements
Stored procedures are identical across all databases	Syntax and capabilities vary by database system

Placement Quick Points

```text
PROCEDURE
→ REUSABLE DATABASE LOGIC

PARAMETERS
→ PROVIDE INPUT

CALL
→ EXECUTE PROCEDURE

PL/pgSQL
→ POSTGRESQL PROCEDURAL LANGUAGE
```

Copy

- A stored procedure is reusable logic stored in the database.
- PostgreSQL procedures are created using `CREATE PROCEDURE`.
- PostgreSQL procedures are executed using `CALL`.
- Procedures can accept parameters and perform multiple operations.
- Procedure syntax and behavior vary between database systems.

Interview Questions

Q1. What is a stored procedure?

A stored procedure is reusable database-side logic stored in the database and executed as a unit.

Q2. How do you execute a PostgreSQL procedure?

Using `CALL`.

`Sql CALL insert_employee('Rahul', 2, 65000);`

Q3. Can a stored procedure accept parameters?

> Yes. Parameters allow the same procedure to be used with different input values.

Q4. Why are stored procedures useful?

> They allow reusable database logic and can perform multiple related operations from a single procedure call.

Q5. What is the difference between a procedure and a function?

> In PostgreSQL, a procedure is invoked with `CALL`, while a function returns a value/result and is commonly invoked in expressions such as `SELECT`.

Practice & Hands-On Exercises

Using the PostgreSQL `employees` and `salaries` tables:

1. Create the `insert_employee` procedure:

```sql
CREATE OR REPLACE PROCEDURE insert_employee(
    p_emp_name VARCHAR,
    p_dept_id INT,
    p_salary NUMERIC
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_emp_id INT;
BEGIN
    INSERT INTO employees (emp_name, dept_id)
    VALUES (p_emp_name, p_dept_id)
    RETURNING emp_id INTO v_emp_id;

    INSERT INTO salaries (emp_id, salary)
    VALUES (v_emp_id, p_salary);
END;
$$;
```

1. Call it with a new employee:

```sql
CALL insert_employee('Rahul', 2, 65000);
```

1. Verify that the employee was inserted:

```sql
SELECT * FROM employees WHERE emp_name = 'Rahul';
```

1. Verify that the salary was inserted:

```sql
SELECT * FROM salaries WHERE emp_id = (SELECT emp_id FROM employees WHERE emp_name = 'Rahul');
```

1. Explain the difference between a stored procedure and a function.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CREATE PROCEDURE
       ↓
ACCEPT PARAMETERS
       ↓
EXECUTE SQL LOGIC
       ↓
CALL PROCEDURE
```

> ***A** PostgreSQL stored procedure is reusable database logic that can accept parameters and execute multiple SQL operations through `CALL`.*

End of Stored Procedures

6.5 Triggers (BEFORE / AFTER)

Level: 6 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **trigger** is a database object that automatically runs when a specified event occurs on a table or view.

Common events are:

```
TEXT

INSERT
UPDATE
DELETE
```

Copy

Two common trigger timings are:

Type	Runs	Common Use
BEFORE	Before the change	Validation, changing values
AFTER	After the change	Logging, auditing

> *Trigger → Automatic database action*

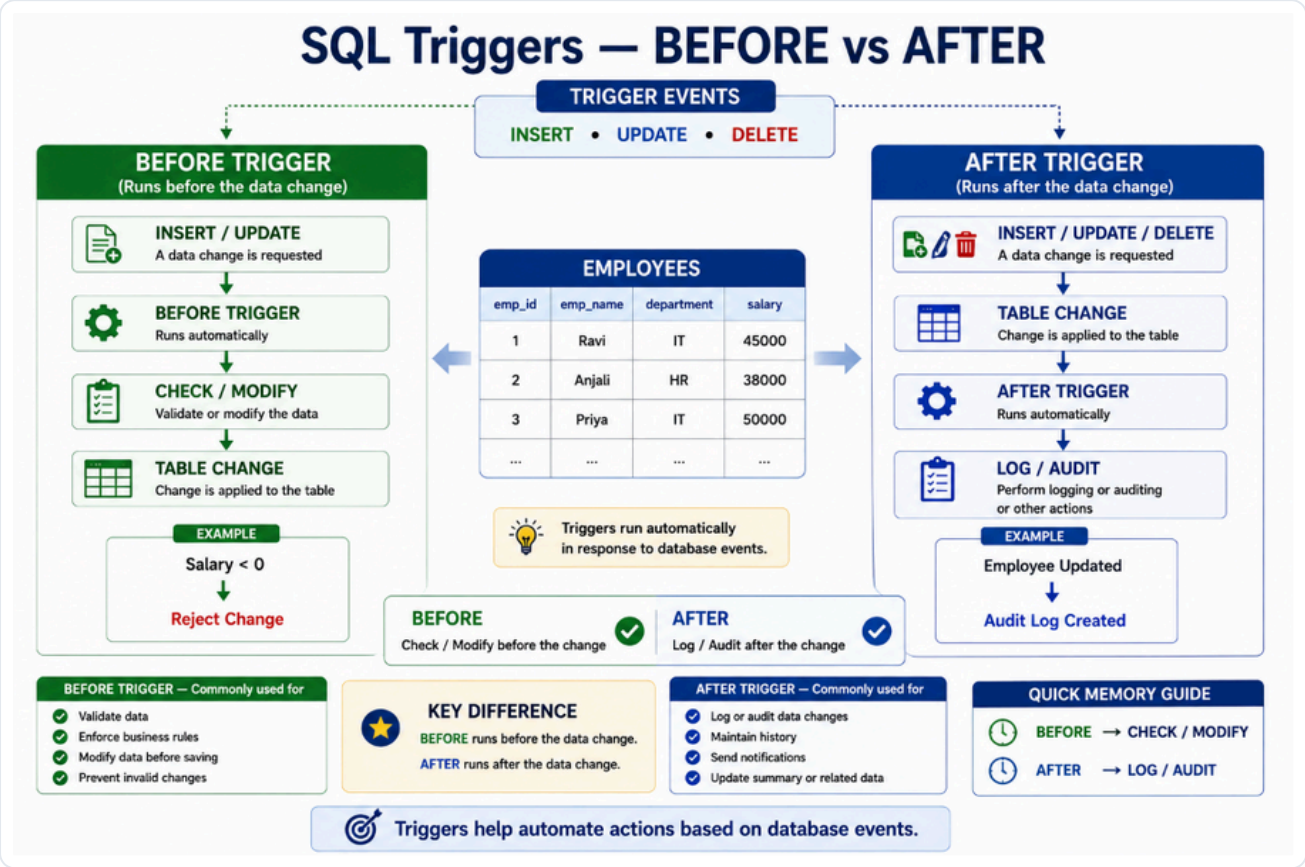


Figure 1: A trigger runs automatically in response to a database event; BEFORE triggers run before the change, while AFTER triggers run after the change.

BEFORE Trigger

A **BEFORE trigger** runs before the database change is completed.

It is commonly used for:

- Validation
- Adjusting values
- Applying rules before insertion or update

PostgreSQL Example

First, create the trigger function:

```sql
CREATE OR REPLACE FUNCTION check_employee()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION 'Salary cannot be negative';
    END IF;

    RETURN NEW;
END;
$$;
```

Then attach it to the table:

```sql
CREATE TRIGGER before_employee_insert
BEFORE INSERT ON employees
FOR EACH ROW
EXECUTE FUNCTION check_employee();
```

Now, an invalid value is rejected before the row is inserted.

> *BEFORE → Validate or modify before the change*

AFTER Trigger

An **AFTER trigger** runs after the data change has successfully occurred.

- Auditing
- Logging
- Recording changes

Example concept:

```
TEXT

UPDATE Employee
      ↓
Table Updated
      ↓
AFTER Trigger
      ↓
Audit Log
```

A PostgreSQL implementation can use a trigger function to record the change in an audit table.

> ***AFTER → Perform an action after the change***

BEFORE vs AFTER

BEFORE Trigger	AFTER Trigger
Runs before the change	Runs after the change
Useful for validation	Useful for auditing/logging
Can modify `NEW` values in applicable row-level triggers	Commonly records or reacts to completed changes

Easy Memory Trick

```
TEXT

BEFORE → CHECK
AFTER  → LOG
```

Important PostgreSQL Terms

For row-level triggers, PostgreSQL provides special values such as:

```
TEXT

NEW → New row value
OLD → Previous row value
```

For example:

```
SQL

NEW.salary
```

refers to the new salary value during a suitable row-level trigger.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
A trigger must be called manually	A trigger fires automatically when its event occurs
BEFORE means before the SQL statement is sent	It runs in response to the database event before the row change is completed
AFTER can validate data before insertion	Validation that must prevent the change is generally done before the change
Triggers are only used for errors	They can also be used for auditing, logging, and automatic actions
A trigger is the same as a function	In PostgreSQL, a trigger uses a trigger function but is a separate database object
Every database uses the same trigger syntax	Trigger syntax and capabilities vary by database system

Placement Quick Points

```text
TRIGGER
→ AUTOMATIC DATABASE ACTION

BEFORE
→ VALIDATE / MODIFY

AFTER
→ LOG / AUDIT
```

- A trigger runs automatically when its defined event occurs.
- Common events are `INSERT`, `UPDATE`, and `DELETE`.
- `BEFORE` triggers run before the row change is completed.
- `AFTER` triggers run after the change.
- PostgreSQL trigger functions are written using languages such as PL/pgSQL.
- `NEW` and `OLD` can provide access to row values in applicable row-level triggers.

Interview Questions

Q1. What is a trigger?

A trigger is a database object that automatically executes in response to a specified database event.

Q2. What is a BEFORE trigger?

A BEFORE trigger runs before the associated row change and is commonly used for validation or modifying values.

Q3. What is an AFTER trigger?

An AFTER trigger runs after the associated change has occurred and is commonly used for logging or auditing.

Q4. What is the difference between a trigger and a trigger function?

The **trigger** defines when the automatic action fires. The **trigger function** contains the code that runs.

Q5. What are NEW and OLD?

For applicable row-level triggers:

```
text NEW → New row values OLD → Previous row values
```

Practice & Hands-On Exercises

Using the `employees` table:

1. Create a PostgreSQL BEFORE trigger that rejects a negative salary:

```TEXT
CREATE OR REPLACE FUNCTION check_employee()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION 'Salary cannot be negative';
    END IF;
    RETURN NEW;
END;
$$;

CREATE TRIGGER before_employee_insert
BEFORE INSERT ON employees
FOR EACH ROW
EXECUTE FUNCTION check_employee();
```

1. Test the trigger with a valid value.
2. Test it with an invalid value.
3. Explain the difference between BEFORE and AFTER triggers.
4. Give one real-world use case for an AFTER trigger.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```SQL
INSERT / UPDATE / DELETE
          ↓
       TRIGGER
       ↙     ↘
   BEFORE    AFTER
     ↓         ↓
 CHECK       LOG / AUDIT
```

> *A trigger automatically runs when a database event occurs; BEFORE triggers are useful for validation, while AFTER triggers are commonly used for logging and auditing.*

End of Triggers (BEFORE / AFTER)

6.6 Users, Roles & Permissions

Level: 6 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Database access is controlled using **users, roles, and permissions**.

- **User** → Account that can connect to the database.
- **Role** → Defines a set of permissions.
- **Permission** → Specifies what a user or role can do.

```text
USER
  ↓
ROLE
  ↓
PERMISSIONS
  ↓
DATABASE OBJECTS
```

> *Users identify who connects; roles and permissions control what they can do.*

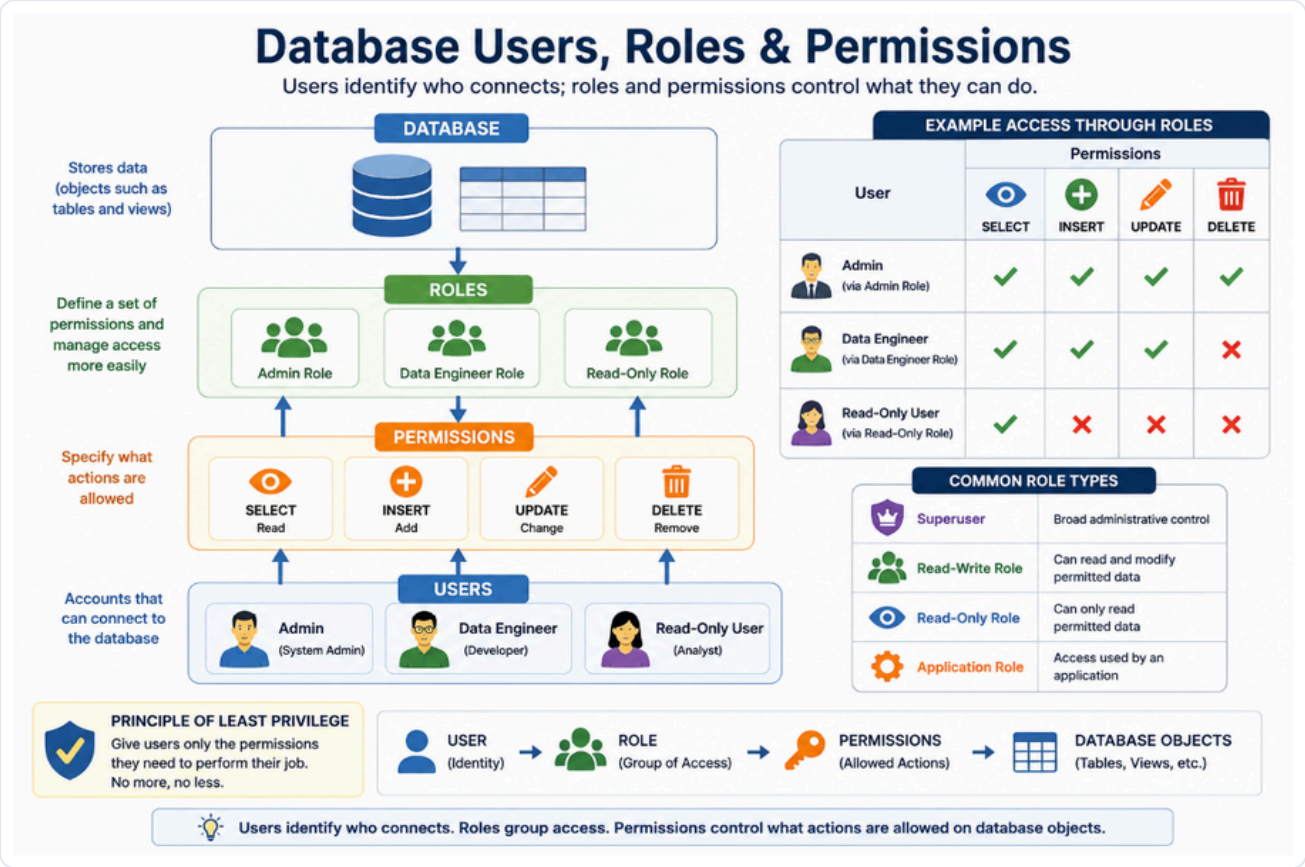


Figure 1: Database users receive permissions directly or through roles that control access to database objects.

User

A **database user** is an account that can authenticate and connect to a database.

For example, in PostgreSQL:

```sql
CREATE USER naaga WITH PASSWORD 'StrongPass123!';
```

A user can then be given specific permissions directly or assigned to roles.

Roles

A **role** is a database identity that can own objects and/or receive permissions. In PostgreSQL, a role can also have the ability to log in.

Roles are useful when multiple users need the same access.

For example:

```
TEXT

HR Role
   ↓
SELECT + INSERT
   ↓
HR Users
```

Copy

This is easier to manage than assigning the same permissions separately to every individual user.

Permissions

Permissions determine what a user or role is allowed to do.

Common permissions include:

```
TEXT

SELECT  → Read
INSERT  → Add
UPDATE  → Change
DELETE  → Remove
```

Copy

Example:

```sql
GRANT SELECT, INSERT
ON employees
TO hr_role;
```

This allows the `hr_role` to perform the granted operations on the `employees` table, subject to the database's permission model.

PostgreSQL Example

A simple access setup can look like:

```sql
CREATE ROLE hr_role;

GRANT SELECT, INSERT
ON employees
TO hr_role;

GRANT hr_role TO naaga;
```

Conceptually:

```text
naaga
  ↓
hr_role
  ↓
SELECT + INSERT
  ↓
employees
```

Common Role Types

Access	Meaning
Superuser	Broad administrative control
Read-Write	Can read and modify permitted data
Read-Only	Can only read permitted data
Application Role	Access used by an application

Note: Exact roles and privileges vary between database systems.

Principle of Least Privilege

Users should receive **only the permissions they need**.

For example:

```text
Student
→ SELECT

Teacher
→ SELECT + UPDATE

Admin
→ Broader administrative permissions
```

This reduces unnecessary access to sensitive data and operations.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
User and role are always completely different objects	In PostgreSQL, both are roles; `LOGIN` distinguishes roles that can connect
Giving everyone full access is easier	Users should receive only required permissions
`SELECT` allows modification	`SELECT` provides read access
A role is the same as a database	A role controls identity/access; a database stores data
Permissions automatically apply everywhere	Permissions must be granted on the relevant objects or inherited through roles

Placement Quick Points

```text
USER
→ ACCOUNT / IDENTITY

ROLE
→ GROUP OF ACCESS RIGHTS

PERMISSION
→ ALLOWED ACTION
```

Copy

- Users/roles identify database identities.
- Permissions control access to database objects.

- Common permissions include `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Roles make permission management easier for groups of users.
- **Least privilege** means giving only the access required.

Interview Questions

Q1. What is a database user?

> A user is an account/role that can authenticate and connect to the database.

Q2. What is a role?

> A role is a database identity that can own objects and/or receive permissions. Roles can also be granted to other roles or users.

Q3. What is a permission?

> A permission defines what operations a user or role can perform on a database object.

Q4. Why are roles useful?

> Roles allow common permissions to be grouped and assigned to multiple users more easily.

Q5. What is the principle of least privilege?

> Give users only the permissions they actually need to perform their required tasks.

Q6. What is the difference between SELECT and UPDATE?

> `text SELECT → Read data UPDATE → Change data`

Practice & Hands-On Exercises

1. Explain the difference between a user, role, and permission.
2. Identify suitable permissions for a student, teacher, and administrator.
3. Explain why roles are useful when many users need the same access.
4. Explain the principle of least privilege.
5. Interpret this command:

```
CODE

GRANT SELECT
ON employees
TO hr_role;
```

Copy

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE

USER
  ↓
ROLE
  ↓
PERMISSIONS
  ↓
DATABASE OBJECT
```

Copy

> ***Users provide identity, roles group access, and permissions control what actions can be performed on database objects.***

End of Users, Roles & Permissions

7.1 DISTINCT and LIMIT

Level: 7 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`DISTINCT` and `LIMIT` help control the rows returned by a `SELECT` query.

Clause	Purpose
`DISTINCT`	Removes duplicate result rows
`LIMIT`	Restricts the number of rows returned

> ***DISTINCT → Remove duplicates***
> ***LIMIT → Limit rows***

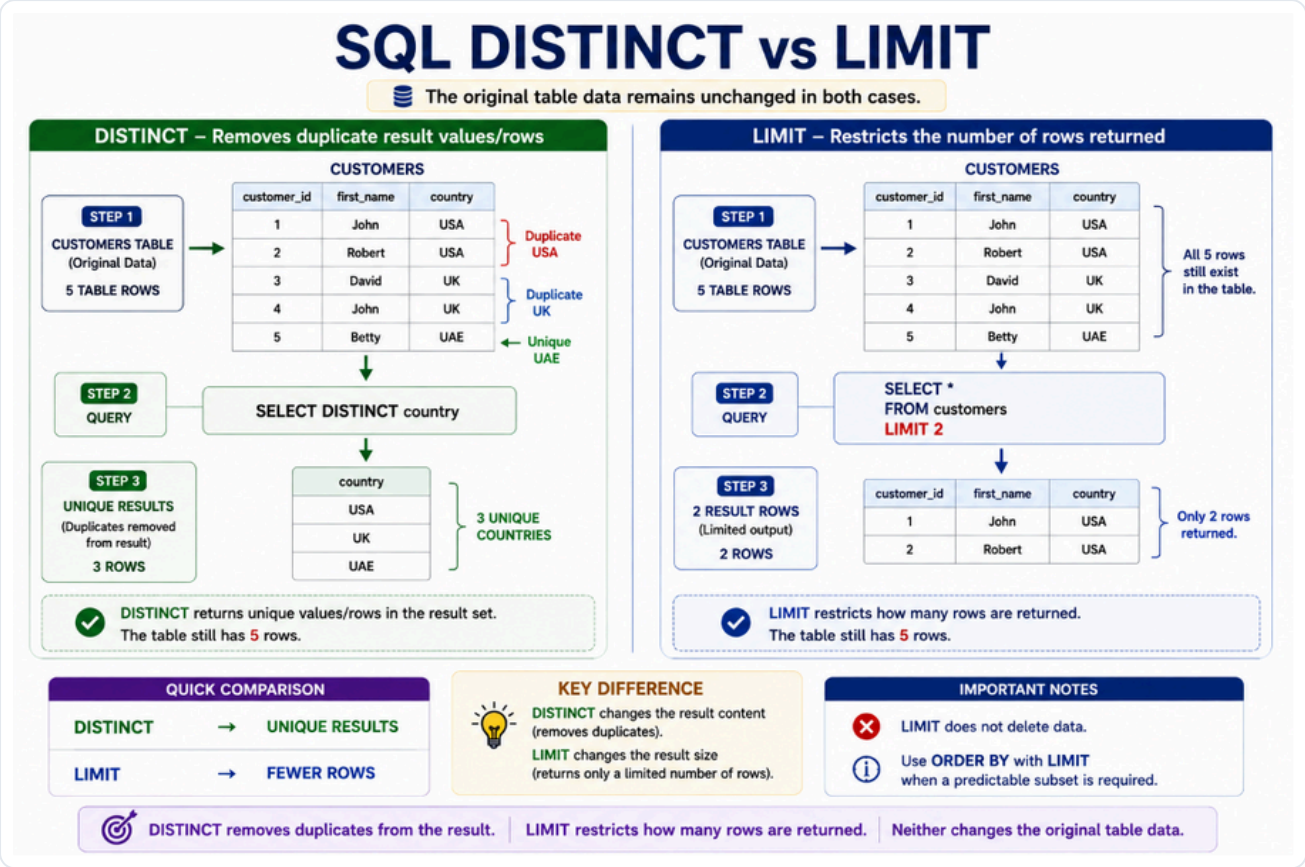


Figure 1: DISTINCT removes duplicate rows from a query result, while LIMIT restricts the number of rows returned.

Practice Table

```sql
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    age INT,
    country VARCHAR(50)
);

INSERT INTO customers
(customer_id, first_name, last_name, age, country)
VALUES
(1, 'John', 'Doe', 31, 'USA'),
(2, 'Robert', 'Luna', 22, 'USA'),
(3, 'David', 'Robinson', 22, 'UK'),
(4, 'John', 'Reinhardt', 25, 'UK'),
(5, 'Betty', 'Doe', 28, 'UAE');
```

DISTINCT

`DISTINCT` removes duplicate rows from the query result.

```sql
SELECT DISTINCT country
FROM customers;
```

Result:

country
USA
UK
UAE

There are 5 customers, but only 3 unique countries.

DISTINCT with Multiple Columns

`DISTINCT` considers the **combination of selected columns**.

```sql
SELECT DISTINCT first_name, country
FROM customers;
```

> ***DISTINCT → Return unique result rows***

LIMIT

`LIMIT` restricts the number of rows returned.

```sql
SELECT *
FROM customers
LIMIT 2;
```

This returns only 2 rows.

> *LIMIT → Return only a specified number of rows*

Important: Without `ORDER BY`, you should not rely on which specific rows are returned as the "first" rows.

DISTINCT vs LIMIT

DISTINCT	LIMIT
Removes duplicate result rows	Restricts result size
Focuses on uniqueness	Focuses on row count
`SELECT DISTINCT country`	`SELECT * ... LIMIT 2`

Easy Memory Trick

```text
DISTINCT → UNIQUE
LIMIT    → COUNT
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`DISTINCT` removes duplicate rows from the table	It removes duplicates from the query result
`DISTINCT` always applies to one column	It applies to the complete selected row/column combination
`LIMIT 2` always returns the same two rows	Without `ORDER BY`, row selection is not guaranteed
`LIMIT` removes data	It only limits the returned result
`DISTINCT` changes the stored table	It only affects the query result

Placement Quick Points

```text
DISTINCT
→ REMOVE DUPLICATE RESULT ROWS

LIMIT
→ RESTRICT NUMBER OF RESULT ROWS
```

- `DISTINCT` returns unique result rows.
- With multiple columns, uniqueness is based on the selected combination.
- `LIMIT` restricts the number of rows returned.
- `LIMIT` does not delete or change stored data.
- Use `ORDER BY` when you need a predictable subset of rows.

Interview Questions

Q1. What is DISTINCT?

`DISTINCT` removes duplicate rows from a query result.

Q2. What is LIMIT?

`LIMIT` restricts the number of rows returned by a query.

Q3. Does DISTINCT delete duplicate data from the table?

No. It only removes duplicates from the result returned by the query.

Q4. What happens with DISTINCT on multiple columns?

The database considers the combination of all selected columns when removing duplicates.

Q5. Why should ORDER BY often be used with LIMIT?

`ORDER BY` makes the selected rows predictable.

Example:

```sql
SELECT * FROM customers ORDER BY customer_id LIMIT 2;
```

Practice & Hands-On Exercises

Using the `customers` table:

1. Display all unique countries:

```sql
SELECT DISTINCT country FROM customers;
```

1. Display unique `first_name` values:

```sql
SELECT DISTINCT first_name FROM customers;
```

1. Return only 2 customers:

```sql
SELECT * FROM customers LIMIT 2;
```

1. Return the first 3 customers ordered by `customer_id`:

```sql
SELECT * FROM customers ORDER BY customer_id LIMIT 3;
```

1. Explain the difference between `DISTINCT` and `LIMIT`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```sql
DISTINCT → UNIQUE RESULTS
LIMIT    → LIMITED ROWS
```

> ***DISTINCT removes duplicate result rows, while LIMIT controls how many rows are returned.***

End of DISTINCT and LIMIT

7.2 SQL Data Types (Numeric, Character, Date/Time, Special)

Level: 7 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **data type** defines what kind of value a column can store.

For example:

```
TEXT

Age       → INT
Price     → DECIMAL
Name      → VARCHAR
Added On  → DATE
In Stock  → BOOLEAN
```

Copy

Choosing the appropriate data type helps maintain **valid data** and makes the database easier to work with.

SQL data types can be broadly grouped into:

1. Numeric
2. Character
3. Date/Time
4. Special

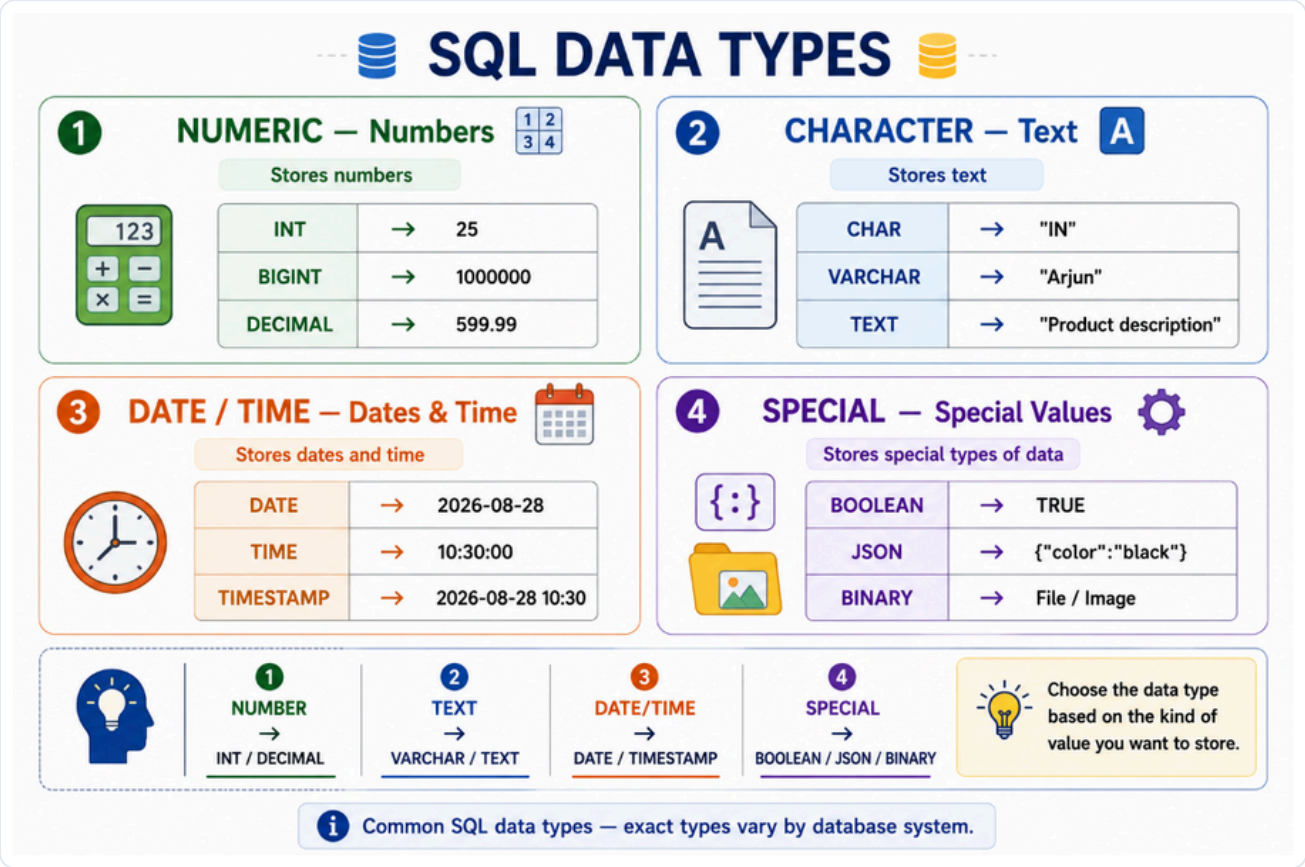


Figure 1: Common SQL data types can be grouped into numeric, character, date/time, and special-purpose types.

1. Numeric Types

Numeric types store **whole numbers or decimal values**.

Type	Common Use
`INT`	Whole numbers
`BIGINT`	Large whole numbers
`DECIMAL(p,s)` / `NUMERIC(p,s)`	Exact decimal values
`FLOAT` / `DOUBLE`	Approximate numeric values

Example:

```
TEXT

Quantity → 10
Price    → 599.99
```

Copy

For values such as money, `DECIMAL`/`NUMERIC` is generally preferred when exact decimal precision is required.

---

2. Character Types

Character types store **text**.

Type	Common Use
`CHAR(n)`	Fixed-length text
`VARCHAR(n)`	Variable-length text
`TEXT`	Longer text

Examples:

```
TEXT

Name        → 'Arjun'
Email       → 'arjun@gmail.com'
Description → 'Wireless mouse'
```

Copy

> *CHAR → Fixed length*
> *VARCHAR → Variable length*

---

3. Date and Time Types

Date/time types store **calendar and time values**.

Type	Stores
`DATE`	Date
`TIME`	Time
`TIMESTAMP`	Date and time

Examples:

```text
2026-08-28
10:30:00
2026-08-28 10:30:00
```

Using proper date/time types is better than storing dates as ordinary text because databases can correctly sort, compare, and calculate with them.

4. Special Types

Some values require special-purpose types.

Type	Common Use
`BOOLEAN`	True/false values
`JSON`	Structured/nested JSON data
Binary types	Images, files, or other binary data

Example:

```text
in_stock → TRUE

metadata → {"color":"black","warranty":1}
```

Note: Exact type names and support vary between database systems.

Practice Table

Example using **PostgreSQL**:

```sql
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    price NUMERIC(10,2),
    in_stock BOOLEAN,
    description TEXT,
    added_on DATE,
    metadata JSONB
);
```

Example row:

```sql
INSERT INTO products
(product_id, product_name, price, in_stock, description, added_on, metadata)
VALUES
(
    1,
    'Wireless Mouse',
    599.99,
    TRUE,
    'Ergonomic wireless mouse',
    '2026-08-28',
    '{"color":"black","warranty_years":1}'
);
```

Here:

```
product_id   → INT
product_name → VARCHAR
price        → NUMERIC
in_stock     → BOOLEAN
description  → TEXT
added_on     → DATE
metadata     → JSONB
```

Choosing the Right Data Type

Choose a type based on the **kind of value**, not just how the value looks.

For example:

```
Age          → INT
Salary       → NUMERIC
Name         → VARCHAR
Birth Date   → DATE
Is Active    → BOOLEAN
```

Avoid storing everything as `VARCHAR`. For example, storing salary as text makes numerical calculations and validation harder.

> *Use the data type that best represents the actual value.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Store every value as `VARCHAR`	Choose a type that matches the data
Store dates as text	Use an appropriate date/time type
Use `FLOAT` for every decimal value	Use `NUMERIC` / `DECIMAL` when exact precision matters
`CHAR` and `VARCHAR` are always identical	They have different length semantics and storage behavior
`BOOLEAN` stores any text	It represents a logical true/false value
All SQL databases support exactly the same types	Data types and syntax vary by DBMS

Placement Quick Points

```
TEXT

NUMERIC
→ INT / BIGINT / DECIMAL

CHARACTER
→ CHAR / VARCHAR / TEXT

DATE/TIME
→ DATE / TIME / TIMESTAMP

SPECIAL
→ BOOLEAN / JSON / BINARY
```

Copy

- A data type defines what kind of value a column can store.
- Numeric types store numbers.
- Character types store text.
- Date/time types store temporal values.
- Special types handle values such as Boolean, JSON, and binary data.
- Choose data types based on the meaning and requirements of the data.
- Exact supported types vary between database systems.

Interview Questions

Q1. What is a SQL data type?

> A data type defines the kind of value a database column can store.

Q2. What is the difference between INT and DECIMAL?

> `INT` stores whole numbers, while `DECIMAL` / `NUMERIC` stores exact decimal values.

Q3. What is the difference between CHAR and VARCHAR?

> `CHAR` is intended for fixed-length character data, while `VARCHAR` is for variable-length character data.

Q4. Why should dates not usually be stored as VARCHAR?

> Proper date/time types allow the database to correctly compare, sort, and perform date calculations.

Q5. Which data type is suitable for true/false values?

> `BOOLEAN`, where supported.

Q6. Which type is commonly used for exact monetary values?

> `DECIMAL` or `NUMERIC`.

Practice & Hands-Hands-On Exercises

Using the `products` table:

1. Identify the data type of each column.
2. Insert a new product:

```sql
INSERT INTO products (product_id, product_name, price, in_stock, description, added_on, metadata)
VALUES (2, 'USB-C Cable', 299.50, TRUE, 'Durable braided cable', '2026-08-28', '{"length":"2m"}');
```

1. Change the product price:

```sql
UPDATE products SET price = 349.99 WHERE product_id = 2;
```

1. Explain why `price` uses `NUMERIC`.
2. Explain why `added_on` uses `DATE`.

3. Give a suitable data type for:
4. Employee age
5. Employee name
6. Joining date
7. Salary
8. Active status

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
TEXT                                                    Copy

NUMBER      → NUMERIC
TEXT        → CHARACTER
DATE/TIME   → TEMPORAL
SPECIAL     → BOOLEAN / JSON / BINARY
```

> ***SQL data types define the kind of values a column stores. Choosing the correct type improves data validity and makes the data easier to query and manage.***

End of SQL Data Types (Numeric, Character, Date/Time, Special)

7.3 CTAS (CREATE TABLE AS SELECT)

Level: 7 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

CTAS stands for **CREATE TABLE AS SELECT**.

It creates a **new table and fills it with the result of a** `SELECT` **query in one statement**.

```
TEXT

SELECT QUERY
     ↓
CREATE NEW TABLE
     ↓
COPY RESULT DATA
```

Copy

> *CTAS → Create a new table from query results*

Important: CTAS copies the selected column definitions and data, but generally does **not** copy the original table's indexes, primary keys, foreign keys, or other constraints. Exact behavior can vary by database system.

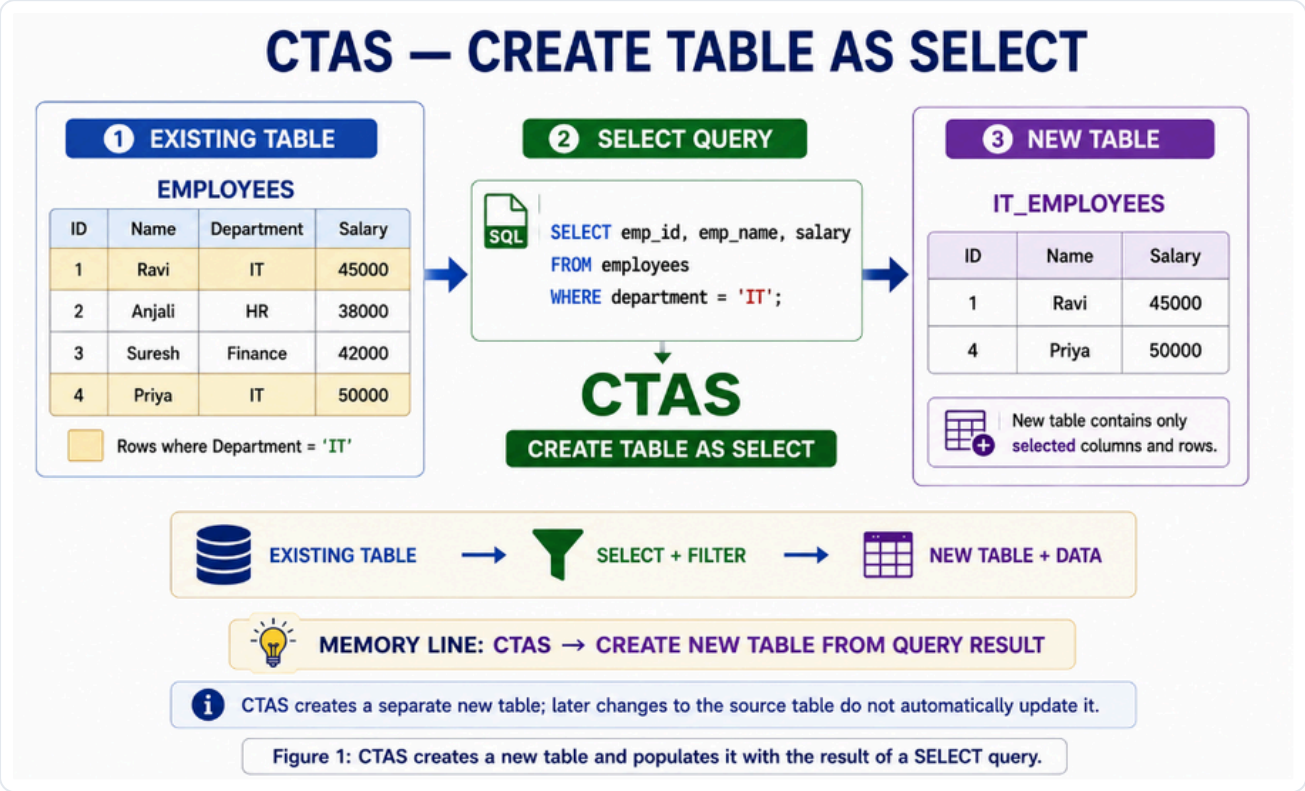


Figure 1: CTAS creates a new table and populates it with the result of a SELECT query.

Syntax

```sql
CREATE TABLE new_table_name AS
SELECT column1, column2
FROM existing_table
WHERE condition;
```

The `SELECT` query determines what data is copied into the new table.

Practice Table

This topic reuses the `employees` table from **5.2 DDL**.

emp_id	emp_name	department	salary
1	Ravi Kumar	IT	45000
2	Anjali Mehta	HR	38000
3	Suresh Rao	Finance	42000
4	Priya Nair	IT	50000

Example: CTAS

Create a new table containing only IT employees:

```sql
CREATE TABLE it_employees AS
SELECT emp_id, emp_name, salary
FROM employees
WHERE department = 'IT';
```

Now query the new table:

```sql
SELECT *
FROM it_employees;
```

Result:

emp_id	emp_name	salary
1	Ravi Kumar	45000
4	Priya Nair	50000

The new table contains the **result of the SELECT query**.

What Does CTAS Copy?

CTAS generally creates the new table based on the selected columns and their resulting data types.

```
TEXT

Original Table
     ↓
SELECT columns + rows
     ↓
New Table
```

Copy

It does **not normally copy** the original table's indexes, primary key, foreign keys, or other constraints. If the new table needs those, they must be created separately.

When Is CTAS Useful?

CTAS is useful when you need to:

- Create a table from query results
- Make a temporary working dataset
- Prepare transformed data

- Create a reporting or analysis table
- Quickly copy selected data

For example:

```sql
CREATE TABLE high_salary_employees AS
SELECT *
FROM employees
WHERE salary > 45000;
```

CTAS vs INSERT INTO

These commands serve different needs:

```text
CTAS
→ CREATE NEW TABLE + INSERT QUERY RESULT

INSERT INTO
→ INSERT DATA INTO AN EXISTING TABLE
```

For example:

```sql
CREATE TABLE it_employees AS
SELECT *
FROM employees
WHERE department = 'IT';
```

Creates the table from scratch.

Whereas:

```sql
INSERT INTO it_employees
SELECT *
FROM employees
WHERE department = 'IT';
```

requires `it_employees` to already exist in the database.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
CTAS modifies the original table	It creates a new table
CTAS only copies data	It creates the new table structure and populates it with the query result
CTAS copies all indexes and constraints	These generally need to be created separately
CTAS and `INSERT INTO` are the same	CTAS creates the table; `INSERT INTO` requires an existing target table
The new table automatically stays synchronized with the source	CTAS creates a separate table; later source changes do not automatically update it

Placement Quick Points

```text
CTAS
→ CREATE TABLE
→ AS SELECT
→ NEW TABLE + QUERY RESULT
```

Copy

- CTAS stands for **CREATE TABLE AS SELECT**.
- It creates a new table from a `SELECT` query result.
- The original table is not modified.
- Selected columns and data are copied into the new table.
- Indexes and constraints generally need to be created separately.
- The new table is independent of later changes to the source table.

Interview Questions

Q1. What is CTAS?

CTAS is a SQL technique that creates a new table and populates it with the result of a `SELECT` query in a single statement.

Q2. Does CTAS modify the source table?

No. It creates a separate new table.

Q3. Does CTAS copy indexes and constraints?

Generally, no. Primary keys, foreign keys, and indexes usually need to be created separately on the new table.

Q4. What is the difference between CTAS and INSERT INTO?

> CTAS creates the target table and fills it with query results. `INSERT INTO` adds data to an existing table.

Q5. Is the new CTAS table automatically updated when the source table changes?

> No. It is a completely separate table and does not automatically stay synchronized with the source table.

Practice & Hands-On Exercises

Using the `employees` table:

1. Create a table containing only IT employees:

```sql
CREATE TABLE it_employees AS
SELECT emp_id, emp_name, salary
FROM employees
WHERE department = 'IT';
```

1. Create a table containing employees with salary greater than `40000`:

```sql
CREATE TABLE high_earners AS
SELECT *
FROM employees
WHERE salary > 40000;
```

1. Select the data from the new table:

```sql
SELECT * FROM high_earners;
```

1. Explain what CTAS creates.

2. Explain the difference between CTAS and `INSERT INTO`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
SELECT
  ↓
CREATE TABLE AS
  ↓
NEW TABLE
  ↓
QUERY RESULT STORED
```

> ***CTAS creates a new table and populates it with the result of a SELECT query in one statement.***

End of CTAS (CREATE TABLE AS SELECT)

7.4 Altering Tables (ADD/DROP/ALTER COLUMN)

Level: 7 | **Version:** v1.0 | **Author:** Meptrasoft

Overview & Core Concepts

Once a table exists, you can still change its structure using `ALTER TABLE`. There are three common alterations:

1. Adding a new column
2. Dropping (removing) a column
3. Changing a column's datatype

Practice Table

This topic reuses the `employees` table from 7.3. If not already created, use the setup script from 7.3.

Adding a Column

```sql
ALTER TABLE table_name
ADD COLUMN column_name datatype;
```

Example:

```sql
ALTER TABLE employees
ADD COLUMN join_date DATE;
```

Dropping a Column

```sql
ALTER TABLE table_name
DROP COLUMN column_name;
```

Example:

```sql
ALTER TABLE employees
DROP COLUMN join_date;
```

Changing a Column's Datatype

```sql
ALTER TABLE table_name
ALTER COLUMN column_name TYPE new_datatype;
```

Example:

```sql
ALTER TABLE employees
ALTER COLUMN emp_name TYPE VARCHAR(100);
```

Placement Quick Points

- Key concept in SQL & Database Engineering: **Altering Tables (ADD/DROP/ALTER COLUMN)**.
- Understanding Altering Tables (ADD/DROP/ALTER COLUMN) is critical for query optimization, schema integrity, and technical placement rounds.
- Always test queries against sample tables to verify edge cases, NULL handling, and expected output.
- Follow standard SQL syntax and indexing best practices for production reliability.

Interview Questions

Q1. What is the core purpose of Altering Tables (ADD/DROP/ALTER COLUMN) in SQL?

> It provides a standardized mechanism to manipulate, define, or query relational database records efficiently while enforcing relational algebra rules.

Q2. What is a common pitfall when working with Altering Tables (ADD/DROP/ALTER COLUMN)?

> Overlooking NULL values, missing indexes on filter/join columns, or writing un-sargable query predicates that cause full table scans.

Q3. How is Altering Tables (ADD/DROP/ALTER COLUMN) evaluated in terms of database performance?

> The query optimizer generates an execution plan; proper usage ensures minimal I/O disk reads and optimal CPU memory utilization.

Practice & Hands-On Exercises

1. Write a query practicing Altering Tables (ADD/DROP/ALTER COLUMN) using the pre-loaded practice tables.
2. Examine the expected output and verify how NULL values or edge cases behave.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

**End of Altering Tables (ADD/DROP/ALTER COLUMN)*

8.1 The WHERE Clause

Level: 8 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

The `WHERE` clause is used to **filter rows** based on a condition.

For example:

```sql
SELECT *
FROM employees
WHERE dept_id = 10;
```

Only employees whose `dept_id` is `10` are returned.

Common filtering operators and clauses include:

```text
WHERE
IN / NOT IN
BETWEEN
IS / IS NOT
LIKE / NOT LIKE
```

These will be covered one by one in this module.

> ***WHERE → Keep only the rows that match a condition.***

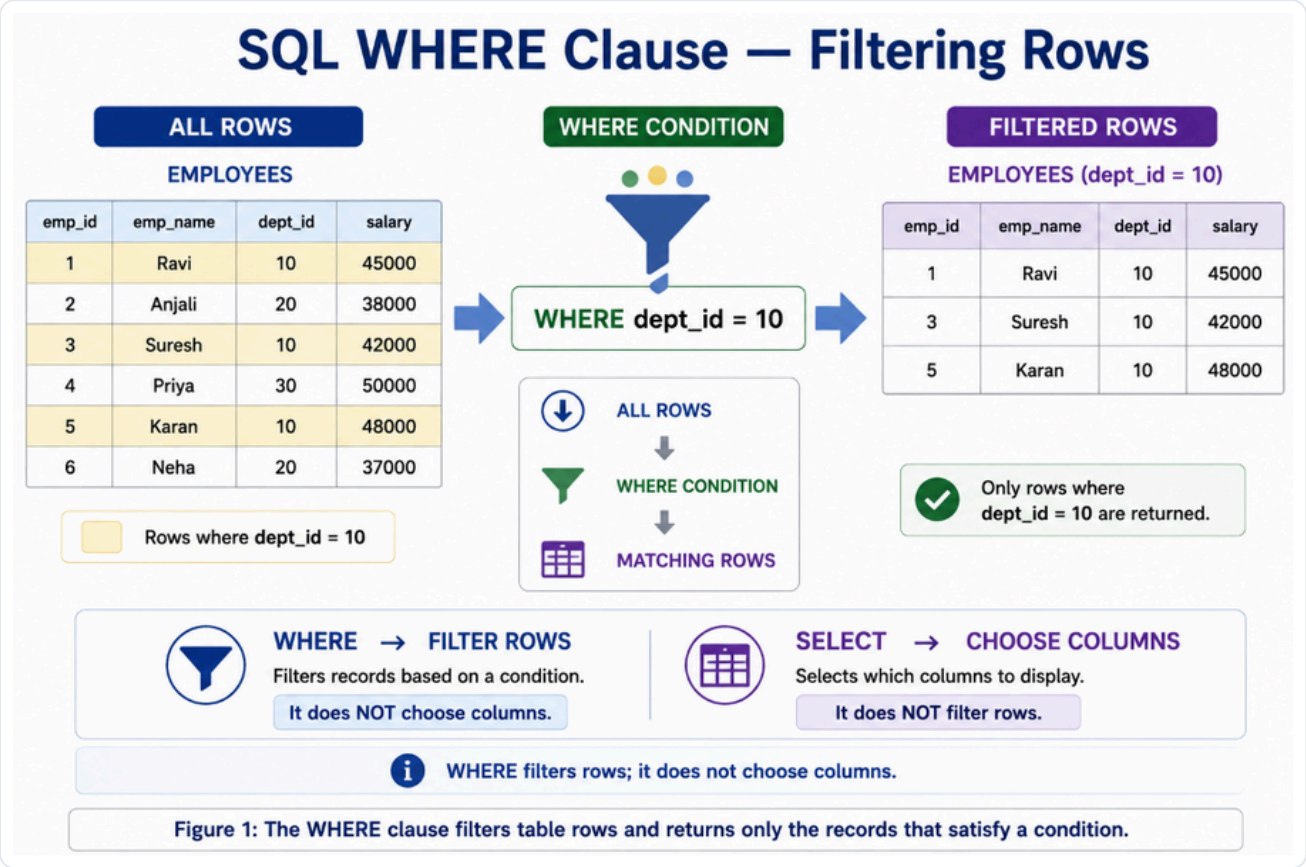


Figure 1: The WHERE clause filters table rows and returns only the records that satisfy a condition.

Practice Table

This topic uses the `employees` table:

emp_id	emp_name	dept_id	salary	phone_no
1	John Smith	10	45000	9876543210
2	Emma Brown	20	60000	NULL
3	Samuel Reed	10	32000	9123456780
4	Anderson	30	55000	NULL
5	Sarah Wilson	20	48000	9988776655
6	Tom Watson	10	51000	9090909090

Basic WHERE

Filter employees from department `10`:

```sql
SELECT *
FROM employees
WHERE dept_id = 10;
```

Result:

emp_id	emp_name	dept_id	salary
1	John Smith	10	45000
3	Samuel Reed	10	32000
6	Tom Watson	10	51000

WHERE with a Numeric Condition

You can also filter numeric values.

```sql
SELECT *
FROM employees
WHERE salary > 50000;
```

Result:

emp_id	emp_name	salary
2	Emma Brown	60000
4	Anderson	55000
6	Tom Watson	51000

Common comparison operators include:

```text
=    → Equal
<>   → Not equal
>    → Greater than
<    → Less than
>=   → Greater than or equal
<=   → Less than or equal
```

WHERE with Text

Text values are usually written inside single quotes:

```sql
SELECT *
FROM employees
WHERE emp_name = 'John Smith';
```

This returns only the row where the name matches exactly.

WHERE with Multiple Conditions

Conditions can be combined using `AND` and `OR`.

AND

Both conditions must be true.

```sql
SELECT *
FROM employees
WHERE dept_id = 10
  AND salary > 40000;
```

OR

At least one condition must be true.

```sql
SELECT *
FROM employees
WHERE dept_id = 10
   OR dept_id = 20;
```

AND → Both conditions
OR → At least one condition

Important: NULL Values

`NULL` means a value is **unknown or missing**.

Do not use:

```sql
WHERE phone_no = NULL
```

Instead, use:

```sql
SELECT *
FROM employees
WHERE phone_no IS NULL;
```

And for non-NULL values:

```sql
SELECT *
FROM employees
WHERE phone_no IS NOT NULL;
```

`IS` and `IS NOT` will be covered in detail later.

WHERE vs SELECT

A common beginner confusion:

```text
SELECT → Which columns?
WHERE  → Which rows?
```

For example:

```sql
SELECT emp_name, salary
FROM employees
WHERE dept_id = 10;
```

Here:

- `SELECT` chooses `emp_name` and `salary`.
- `WHERE` keeps only employees from department `10`.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`WHERE` chooses columns	`WHERE` filters rows
Text values do not need quotes	String literals are normally written using single quotes
`= NULL` checks for NULL	Use `IS NULL` or `IS NOT NULL`
`AND` means either condition can be true	Both conditions must be true
`OR` means both conditions must be true	At least one condition must be true
`WHERE` permanently removes rows	It only filters the query result

Placement Quick Points

```text
WHERE
→ FILTER ROWS

=
→ EQUAL

>
→ GREATER THAN

AND
→ BOTH CONDITIONS

OR
→ AT LEAST ONE CONDITION

IS NULL
→ CHECK FOR NULL
```

Copy

- `WHERE` filters rows based on conditions.
- Comparison operators can be used inside `WHERE`.
- Text values are normally enclosed in single quotes.
- `AND` requires all combined conditions to be true.
- `OR` requires at least one condition to be true.
- Use `IS NULL` and `IS NOT NULL` for `NULL` checks.
- `WHERE` does not change the stored table.

Interview Questions

Q1. What is the purpose of WHERE?

> `WHERE` filters rows based on a specified condition.

Q2. What is the difference between SELECT and WHERE?

`text SELECT → Chooses columns WHERE → Filters rows`

Q3. How do you check for NULL?

`sql WHERE phone_no IS NULL;`

Q4. What is the difference between AND and OR?

`AND` requires all conditions to be true, while `OR` requires at least one condition to be true.

Q5. Does WHERE modify the data?

No. It only filters the rows returned by the query.

Q6. Can WHERE be used with text and numbers?

Yes.

`Sql WHERE department = 'IT'`

and:

`Sql WHERE salary > 50000`

Practice & Hands-On Exercises

Using the `employees` table:

1. Find employees from department `10`:

```
CODE                                                    Copy

SELECT * FROM employees WHERE dept_id = 10;
```

1. Find employees with salary greater than `40000`:

```sql
SELECT * FROM employees WHERE salary > 40000;
```

1. Find the employee named `John Smith`:

```sql
SELECT * FROM employees WHERE emp_name = 'John Smith';
```

1. Find employees in department `10` with salary greater than `40000`:

```sql
SELECT * FROM employees WHERE dept_id = 10 AND salary > 40000;
```

1. Find employees in department `10` or `20`:

```sql
SELECT * FROM employees WHERE dept_id = 10 OR dept_id = 20;
```

1. Find employees whose `phone_no` is `NULL`:

```sql
SELECT * FROM employees WHERE phone_no IS NULL;
```

1. Explain the difference between `SELECT` and `WHERE`.
2. Explain the difference between `AND` and `OR`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```SQL
SELECT → CHOOSE COLUMNS
WHERE  → FILTER ROWS
```

> *The WHERE clause filters rows by applying conditions, allowing SQL to return only the data that matches the requirement.*

End of The WHERE Clause

8.2 Comparison Operators

Level: 8 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Comparison operators are used to compare values, usually inside a `WHERE` clause.

They help SQL determine whether a condition is true or false.

Operator	Meaning
`=`	Equal to
`!=` / `<>`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

> *Comparison operator → Compare values and filter rows*

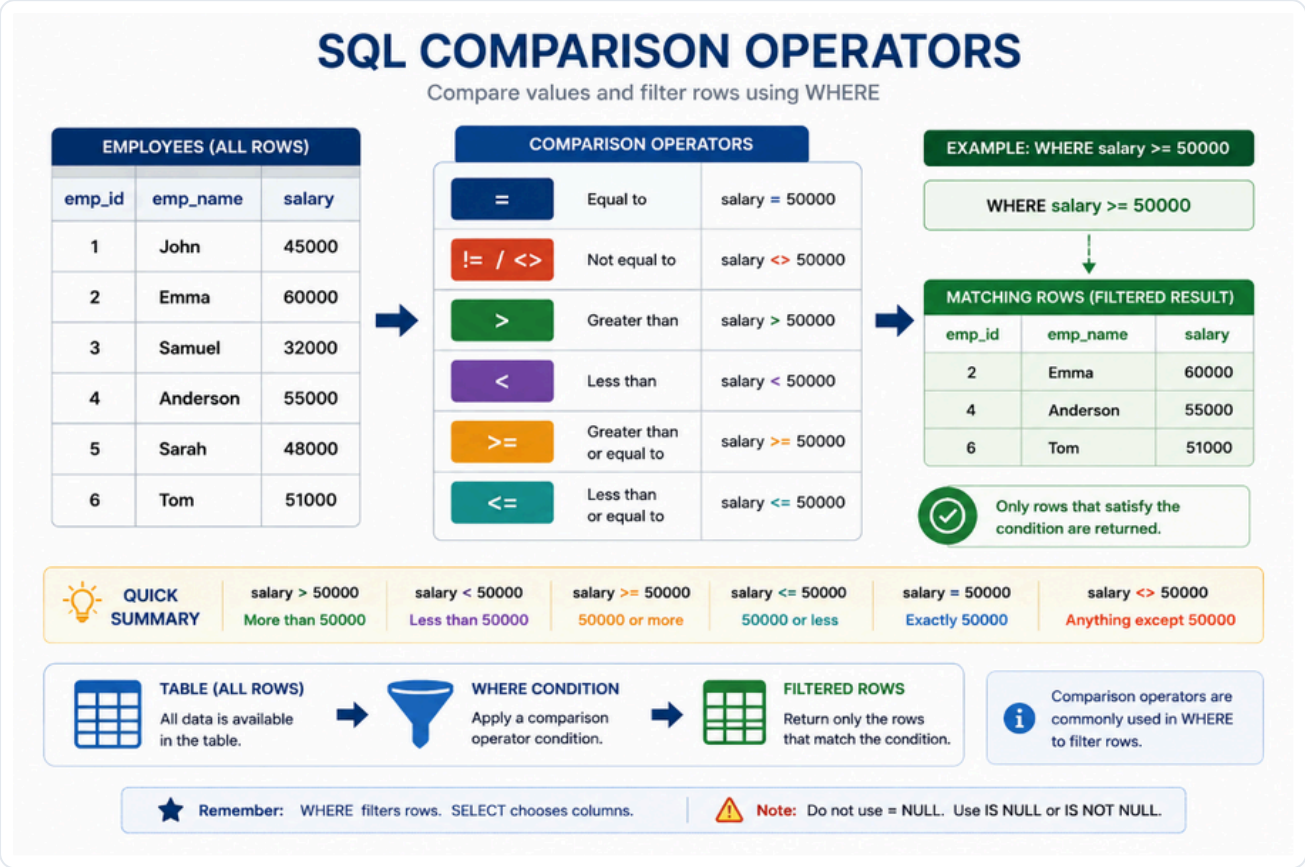


Figure 1: Comparison operators compare column values with conditions and are commonly used with WHERE to filter rows.

Practice Table

This topic reuses the `employees` table from **8.1 The WHERE Clause**.

emp_id	emp_name	dept_id	salary	phone_no
1	John Smith	10	45000	9876543210
2	Emma Brown	20	60000	NULL
3	Samuel Reed	10	32000	9123456780
4	Anderson	30	55000	NULL
5	Sarah Wilson	20	48000	9988776655
6	Tom Watson	10	51000	9090909090

Examples

Equal `=`

```sql
SELECT *
FROM employees
WHERE dept_id = 10;
```

Returns employees whose department is `10`.

Not Equal `!=` or `<>`

```sql
SELECT *
FROM employees
WHERE dept_id != 10;
```

Returns employees whose department is not `10`.

`!=` and `<>` are commonly supported for "not equal," although `<>` is the standard SQL operator.

Greater Than `>`

```sql
SELECT *
FROM employees
WHERE salary > 50000;
```

Returns employees whose salary is greater than `50000`.

Less Than `<`

```sql
SELECT *
FROM employees
WHERE salary < 50000;
```

Returns employees whose salary is less than `50000`.

Greater Than or Equal `>=`

```sql
SELECT *
FROM employees
WHERE salary >= 50000;
```

Returns employees whose salary is `50000` or more.

Less Than or Equal `<=`

```sql
SELECT *
FROM employees
WHERE salary <= 50000;
```

Returns employees whose salary is `50000` or less.

Quick Comparison

```text
salary > 50000   → More than 50000
salary < 50000   → Less than 50000
salary >= 50000  → 50000 or more
salary <= 50000  → 50000 or less
salary = 50000   → Exactly 50000
salary <> 50000  → Anything except 50000
```

> *The operator determines exactly which values satisfy the condition.*

Important: NULL Values

Comparison operators do not work with `NULL` in the same way as normal values.

For example:

```sql
WHERE phone_no = NULL
```

is not the correct way to test for a missing value.

Use:

```sql
WHERE phone_no IS NULL;
```

The `IS NULL` topic will be covered separately.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`=` means assignment in SQL	In a `WHERE` condition, `=` means equal
`!=` is the only not-equal operator	`<>` is also commonly used
`>=` means only greater than	It includes equality
`<=` means only less than	It includes equality
`= NULL` checks for NULL	Use `IS NULL`
Comparison operators change the data	They normally filter or compare values in a query

Placement Quick Points

```text
=    → EQUAL
<>   → NOT EQUAL
>    → GREATER
<    → LESS
>=   → GREATER OR EQUAL
<=   → LESS OR EQUAL
```

- Comparison operators are commonly used inside `WHERE`.
- They compare values and produce a condition used for filtering.
- `=` checks equality.
- `<>` and `!=` commonly mean not equal.
- `>=` and `<=` include equality.
- `NULL` should be checked using `IS NULL` or `IS NOT NULL`.

Interview Questions

Q1. What are comparison operators in SQL?

> They are operators used to compare values in conditions.

Q2. What is the difference between `=` and `>=`?

> ```text = → Exactly equal
>
> = → *Greater than or equal* ```

Q3. What is the not-equal operator in SQL?

`<>` is the standard SQL not-equal operator, and `!=` is also commonly supported.

Q4. Can comparison operators be used with WHERE?

Yes. They are commonly used with `WHERE` to filter rows.

Q5. How do you compare a value with NULL?

Use:

`Sql WHERE phone_no IS NULL;`

not:

`Sql WHERE phone_no = NULL;`

Q6. What is the difference between `>` and `>=`?

`>` excludes the boundary value, while `>=` includes it.

Practice & Hands-On Exercises

Using the `employees` table:

1. Find employees with `dept_id = 10`:

```
CODE
Copy
SELECT * FROM employees WHERE dept_id = 10;
```

1. Find employees with `salary > 50000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary > 50000;
```

1. Find employees with `salary < 50000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary < 50000;
```

1. Find employees with `salary >= 50000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary >= 50000;
```

1. Find employees with `salary <= 50000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary <= 50000;
```

1. Find employees whose department is not `10`:

```
CODE
Copy
SELECT * FROM employees WHERE dept_id <> 10;
```

1. Explain the difference between `>` and `>=`.
2. Explain why `= NULL` should not be used.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
=    → EQUAL
<>   → NOT EQUAL
>    → GREATER
<    → LESS
>=   → GREATER OR EQUAL
<=   → LESS OR EQUAL
```

Comparison operators allow SQL to compare values and filter rows based on conditions.

End of Comparison Operators

8.3 Logical Operators (AND, OR, NOT)

Level: 8 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Logical operators are used to **combine or reverse conditions** in a `WHERE` clause.

Operator	Meaning	Example
`AND`	All conditions must be true	`salary > 30000 AND dept_id = 10`
`OR`	At least one condition must be true	`dept_id = 10 OR dept_id = 20`
`NOT`	Reverses a condition	`NOT dept_id = 10`

> ***AND → Both***
> ***OR → Either***
> ***NOT → Reverse***

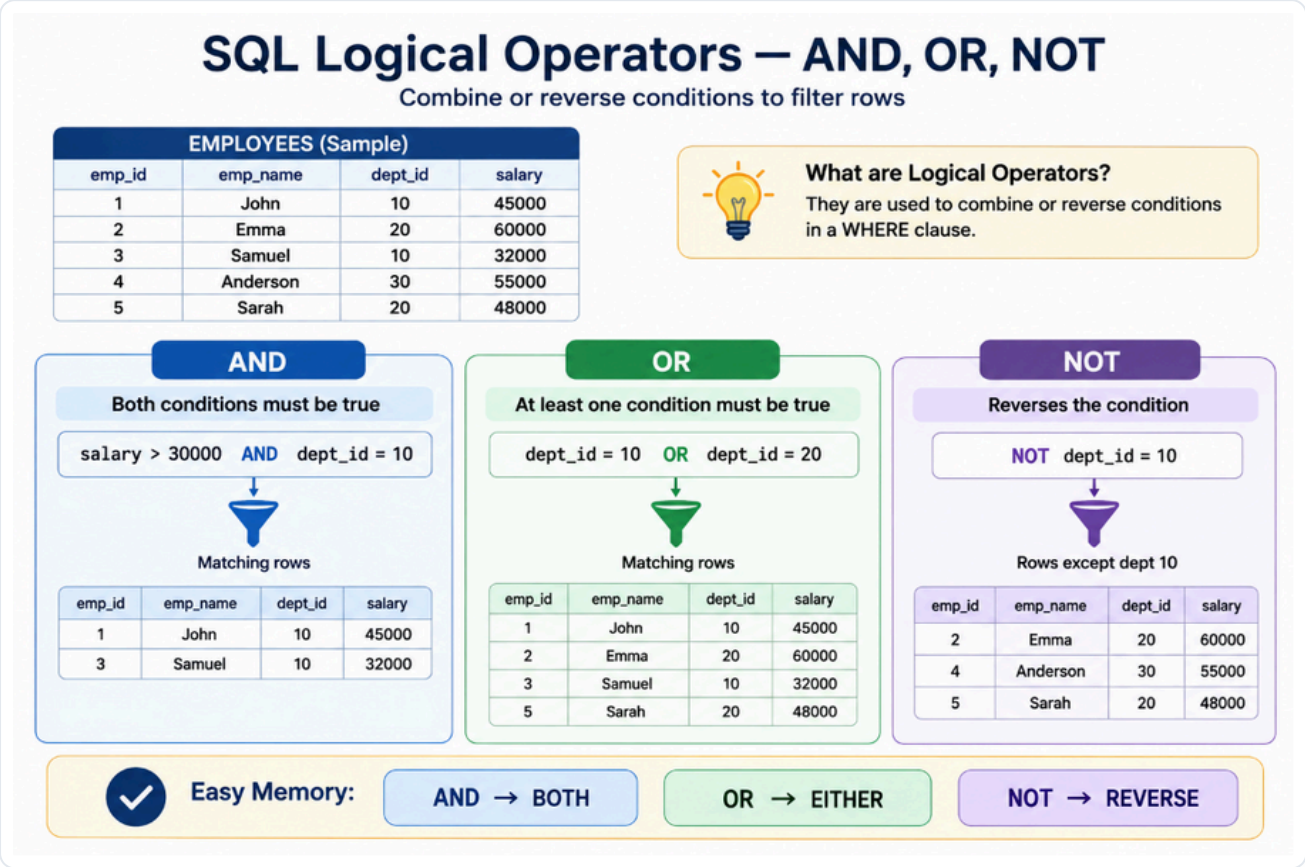


Figure 1: AND, OR, and NOT combine or reverse conditions used to filter rows.

Practice Table

This topic reuses the `employees` table from **8.1 The WHERE Clause**.

emp_id	emp_name	dept_id	salary
1	John Smith	10	45000
2	Emma Brown	20	60000
3	Samuel Reed	10	32000
4	Anderson	30	55000
5	Sarah Wilson	20	48000
6	Tom Watson	10	51000

AND

`AND` is used when **all conditions must be true**.

```sql
SELECT *
FROM employees
WHERE salary > 30000
  AND dept_id = 10;
```

This returns employees who:

- belong to department `10`
- and have a salary greater than `30000`

> ***AND → All conditions must be true***

OR

`OR` is used when **at least one condition must be true**.

```sql
SELECT *
FROM employees
WHERE dept_id = 10
   OR dept_id = 20;
```

This returns employees from department `10` or `20`.

> ***OR → At least one condition must be true***

NOT

`NOT` reverses a condition.

```sql
SELECT *
FROM employees
WHERE NOT dept_id = 10;
```

This returns employees whose department is not `10`.

A common equivalent form is:

> *NOT → Reverse the condition*

Combining Operators

Logical operators can be combined.

```sql
SELECT *
FROM employees
WHERE dept_id = 10
  AND salary > 40000;
```

You can also use parentheses to make the intended logic clear:

```sql
SELECT *
FROM employees
WHERE (dept_id = 10 OR dept_id = 20)
  AND salary > 45000;
```

Here, the department must be `10` or `20`, **and** the salary must be greater than `45000`.

> *Use parentheses when combining multiple AND/OR conditions to make the logic explicit.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`AND` means either condition can be true	All conditions must be true
`OR` means all conditions must be true	At least one condition must be true
`NOT` removes rows permanently	It only reverses the condition in the query
Complex AND/OR conditions need no parentheses	Parentheses can make the intended logic clear and avoid mistakes
Logical operators change stored data	They normally filter the query result

Placement Quick Points

```text
AND → ALL CONDITIONS
OR  → AT LEAST ONE
NOT → REVERSE
```

- `AND` combines conditions that must all be true.
- `OR` combines conditions where at least one must be true.
- `NOT` reverses a condition.
- Parentheses are useful when combining multiple `AND` and `OR` conditions.
- Logical operators are commonly used with `WHERE`.

Interview Questions

Q1. What are logical operators in SQL?

Logical operators combine or reverse conditions used in SQL expressions.

Q2. What is the difference between AND and OR?

```
text AND → All conditions must be true OR → At least one condition must be true
```

Q3. What does NOT do?

`NOT` reverses the result of a condition.

Q4. Why are parentheses useful in logical conditions?

They make the intended grouping of conditions clear, especially when `AND` and `OR` are used together.

Q5. Can AND, OR, and NOT be combined?

Yes.

```
Sql SELECT * FROM employees WHERE (dept_id = 10 OR dept_id = 20) AND salary > 45000;
```

Practice & Hands-On Exercises

Using the `employees` table:

1. Find employees from department `10` **and** with salary greater than `40000`:

```
CODE
Copy

SELECT * FROM employees WHERE dept_id = 10 AND salary > 40000;
```

1. Find employees from department `10` **or** `20`:

```
CODE
Copy

SELECT * FROM employees WHERE dept_id = 10 OR dept_id = 20;
```

1. Find employees not belonging to department `10`:

```
CODE
Copy

SELECT * FROM employees WHERE NOT dept_id = 10;
```

1. Find employees from department `10` or `20` with salary greater than `45000`:

```
CODE
Copy

SELECT * FROM employees WHERE (dept_id = 10 OR dept_id = 20) AND salary > 45000;
```

1. Explain the difference between `AND`, `OR`, and `NOT`.
2. Explain why parentheses are useful in combined conditions.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE
Copy

AND → BOTH
OR  → EITHER
NOT → REVERSE
```

> *Logical operators combine conditions in SQL: AND requires all conditions, OR requires at least one, and NOT reverses a condition.*

End of Logical Operators (AND, OR, NOT)

8.4 IN / NOT IN

Level: 8 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

The `IN` operator allows you to easily specify **multiple values in a `WHERE` clause**.

Instead of chaining multiple `OR` conditions, `IN` provides a clean and compact way to match any value against a list.

Operator	Purpose	Equivalent Condition
`IN`	Matches any value in a specified list	Chained `OR` (`col = A OR col = B`)
`NOT IN`	Excludes all values in a specified list	Chained `AND !=` (`col != A AND col != B`)

> ***IN → Matches ANY value in the list***
> ***NOT IN → Excludes ALL values in the list***

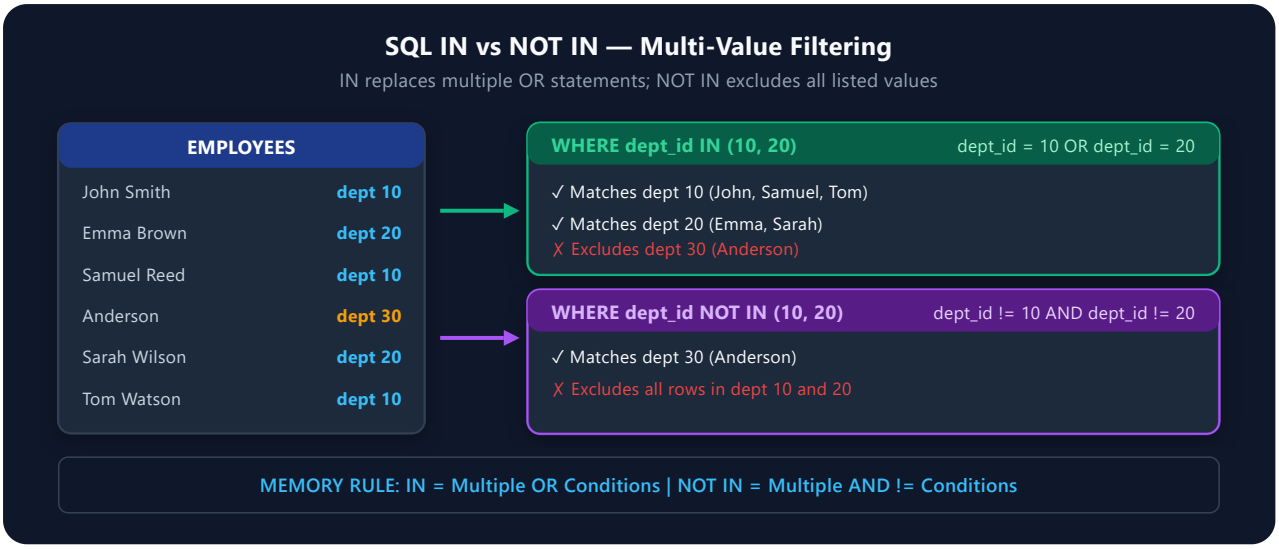


Figure 1: IN matches any value specified in a list, while NOT IN excludes all values in the list.

Practice Table

This topic reuses the `employees` table from **8.1 The WHERE Clause**.

emp_id	emp_name	dept_id	salary
1	John Smith	10	45000
2	Emma Brown	20	60000
3	Samuel Reed	10	32000
4	Anderson	30	55000
5	Sarah Wilson	20	48000
6	Tom Watson	10	51000

Using IN

Instead of writing verbose multiple `OR` statements:

```sql
-- Long syntax:
SELECT *
FROM employees
WHERE dept_id = 10 OR dept_id = 20;
```

You can write concise SQL using `IN`:

```sql
-- Clean syntax with IN:
SELECT *
FROM employees
WHERE dept_id IN (10, 20);
```

This returns all employees belonging to department 10 OR 20:

```text
emp_id | emp_name     | dept_id | salary
1      | John Smith   | 10      | 45000
2      | Emma Brown   | 20      | 60000
3      | Samuel Reed  | 10      | 32000
5      | Sarah Wilson | 20      | 48000
6      | Tom Watson   | 10      | 51000
```

Using NOT IN

`NOT IN` returns rows where the column value is **not** present in the specified list:

```sql
SELECT *
FROM employees
WHERE dept_id NOT IN (10, 20);
```

This filters out all employees in departments 10 and 20, returning only department 30:

```
emp_id | emp_name | dept_id | salary
4      | Anderson | 30      | 55000
```

> ***NOT IN Equivalent:*** `dept_id != 10 AND dept_id != 20`

Critical Caveat: NOT IN with NULL Values

> *[!WARNING] If the list in `NOT IN` contains a `NULL`, the entire expression evaluates to `UNKNOWN`, and the query returns **zero rows**!*

```sql
-- Returns ZERO rows because (10, NULL) causes UNKNOWN for every comparison:
SELECT * FROM employees WHERE dept_id NOT IN (10, NULL);
```

Why? Because `dept_id != 10 AND dept_id != NULL`. Since `!= NULL` is always `UNKNOWN`, `TRUE AND UNKNOWN` produces `UNKNOWN`, which fails the `WHERE` condition.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Writing `WHERE dept_id = (10, 20)`	Use `IN (10, 20)` when comparing against multiple values
Assuming `NOT IN` works like `OR`	`NOT IN (10, 20)` means `!= 10 AND != 20`
Including `NULL` inside `NOT IN (...)`	A single NULL in `NOT IN` invalidates the entire filter
Using string quotes for integers (`IN ('10', '20')`)	Match data types; numbers should not be enclosed in quotes

Placement Quick Points

```text
IN (A, B, C)
→ col = A OR col = B OR col = C

NOT IN (A, B, C)
→ col != A AND col != B AND col != C
```

- `IN` tests membership in a set of values or subquery results.
- `NOT IN` excludes all values present in the set.
- Much cleaner and more readable than multiple chained `OR` / `AND` statements.
- Beware of `NULL` in `NOT IN` lists, which causes queries to return zero rows.

Interview Questions

Q1. What does the IN operator do in SQL?

`IN` determines whether a specified value matches any value in a list or subquery.

Q2. How is IN related to the OR operator?

`column IN (val1, val2)` is shorthand syntactic sugar for `column = val1 OR column = val2`.

Q3. How does NOT IN behave compared to AND?

`column NOT IN (val1, val2)` expands to `column != val1 AND column != val2`.

Q4. What happens if a NOT IN list contains a NULL value?

The condition evaluates to `UNKNOWN` for all rows, resulting in an empty query result set (zero rows returned).

Q5. Can IN be used with subqueries?

Yes! `WHERE dept_id IN (SELECT dept_id FROM departments WHERE budget > 100000)` is standard SQL.

Practice & Hands-On Exercises

Using the `employees` table:

1. Find all employees who work in department 10 or 30:

```sql
SELECT * FROM employees WHERE dept_id IN (10, 30);
```

1. Find all employees who do not work in department 10:

```sql
SELECT * FROM employees WHERE dept_id NOT IN (10);
```

1. Rewrite `emp_name = 'John Smith' OR emp_name = 'Emma Brown'` using `IN`.
2. Test what happens when you run `SELECT * FROM employees WHERE dept_id NOT IN (10, NULL);`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
IN     → MATCH ANY (OR)
NOT IN → EXCLUDE ALL (AND !=)
```

> ***Use IN to match against multiple candidate values cleanly, and ensure NOT IN lists do not contain NULL values.***

End of IN / NOT IN

8.5 BETWEEN

Level: 8 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`BETWEEN` is used to filter values within a **range**.

The starting and ending values are both included.

```SQL
SELECT *
FROM employees
WHERE salary BETWEEN 20000 AND 50000;
```

This returns employees whose salary is:

```TEXT
>= 20000
AND
<= 50000
```

> ***BETWEEN → Check whether a value falls within a range.***

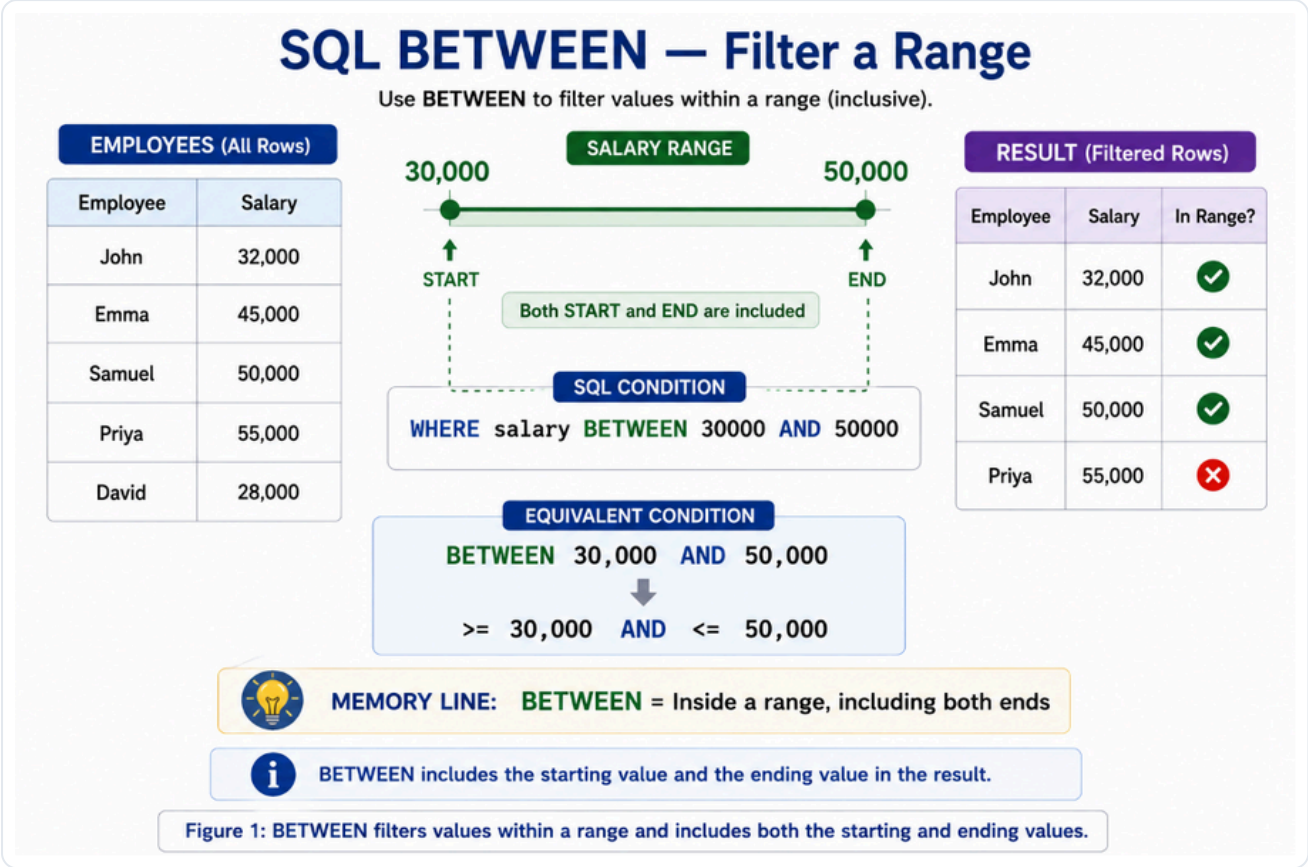


Figure 1: BETWEEN filters values within an inclusive range, including both the starting and ending values.

Practice Table

This topic reuses the `employees` table from **8.1 The WHERE Clause**.

emp_id	emp_name	dept_id	salary
1	John Smith	10	45000
2	Emma Brown	20	60000
3	Samuel Reed	10	32000
4	Anderson	30	55000
5	Sarah Wilson	20	48000
6	Tom Watson	10	51000

BETWEEN with Numbers

Find employees whose salary is between `30000` and `50000`:

```sql
SELECT *
FROM employees
WHERE salary BETWEEN 30000 AND 50000;
```

This includes salaries of exactly `30000` and `50000`.

> *BETWEEN is inclusive.*

BETWEEN with Dates

`BETWEEN` can also be used with dates.

For example:

```sql
SELECT *
FROM employees
WHERE join_date BETWEEN '2026-01-01' AND '2026-06-30';
```

This selects dates within the specified range, including the boundary dates.

BETWEEN vs Comparison Operators

This:

```sql
WHERE salary BETWEEN 30000 AND 50000
```

is equivalent to:

```sql
WHERE salary >= 30000
  AND salary <= 50000
```

`BETWEEN` is often easier to read when working with ranges.

NOT BETWEEN

`NOT BETWEEN` can be used to exclude a range.

```sql
SELECT *
FROM employees
WHERE salary NOT BETWEEN 30000 AND 50000;
```

This returns salaries below `30000` or above `50000`.

> ***NOT BETWEEN → Outside the specified range.***

NULL Values

If the value being tested is `NULL`, the `BETWEEN` condition does not evaluate to `TRUE`.

For example:

```text
salary = NULL
```

does not match:

```sql
WHERE salary BETWEEN 30000 AND 50000
```

`NULL` requires separate handling with `IS NULL` or `IS NOT NULL`.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`BETWEEN` excludes the boundary values	Both boundary values are included
`BETWEEN 30000 AND 50000` means only values strictly inside	`30000` and `50000` are included
`BETWEEN` works only with numbers	It can also be used with comparable values such as dates
`BETWEEN` changes the stored data	It only filters query results
`NOT BETWEEN` means equal to the boundaries	It means outside the specified range

Placement Quick Points

```text
BETWEEN
→ RANGE FILTER

START
→ INCLUDED

END
→ INCLUDED

NOT BETWEEN
→ OUTSIDE THE RANGE
```

- `BETWEEN` filters values within a range.
- The start and end values are included.

- `BETWEEN` can be used with numbers, dates, and other comparable values.
- `NOT BETWEEN` excludes the specified range.
- `NULL` values require separate handling.

Interview Questions

Q1. What is BETWEEN in SQL?

`BETWEEN` filters values that fall within a specified range.

Q2. Is BETWEEN inclusive?

Yes. Both the lower and upper boundary values are included.

Q3. What is the equivalent of BETWEEN?

```sql
salary BETWEEN 30000 AND 50000
```

is equivalent to:

```sql
salary >= 30000 AND salary <= 50000
```

Q4. Can BETWEEN be used with dates?

Yes.

```sql
WHERE join_date BETWEEN '2026-01-01' AND '2026-06-30'
```

Q5. What is NOT BETWEEN?

It selects values outside the specified range.

Q6. Does BETWEEN include NULL?

No. A `NULL` value does not satisfy a `BETWEEN` condition.

Practice & Hands-On Exercises

Using the `employees` table:

1. Find employees with salary between `30000` and `50000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary BETWEEN 30000 AND 50000;
```

1. Find employees with salary between `40000` and `60000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary BETWEEN 40000 AND 60000;
```

1. Find employees whose salary is not between `30000` and `50000`:

```
CODE
Copy
SELECT * FROM employees WHERE salary NOT BETWEEN 30000 AND 50000;
```

1. Write the equivalent `>=` and `<=` condition for a `BETWEEN` query:

```
CODE
Copy
SELECT * FROM employees WHERE salary >= 30000 AND salary <= 50000;
```

1. Explain why `BETWEEN` includes both boundary values.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE Copy

```
BETWEEN 30000 AND 50000
        ↓
>= 30000 AND <= 50000
```

BETWEEN is used to filter values within an inclusive range, including both the starting and ending values.

End of BETWEEN

8.6 IS NULL / IS NOT NULL

Level: 8 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

In SQL, `NULL` represents **missing, unknown, or non-existent data**.

Because `NULL` is not a regular value, it cannot be compared using normal operators like `=` or `!=`. Instead, SQL provides specialized operators:

Operator	Purpose	Example
`IS NULL`	Finds rows where the value is missing / unknown	`phone_no IS NULL`
`IS NOT NULL`	Finds rows where a value exists	`phone_no IS NOT NULL`

> ***IS NULL → Find missing or unknown values***
> ***IS NOT NULL → Find present or recorded values***

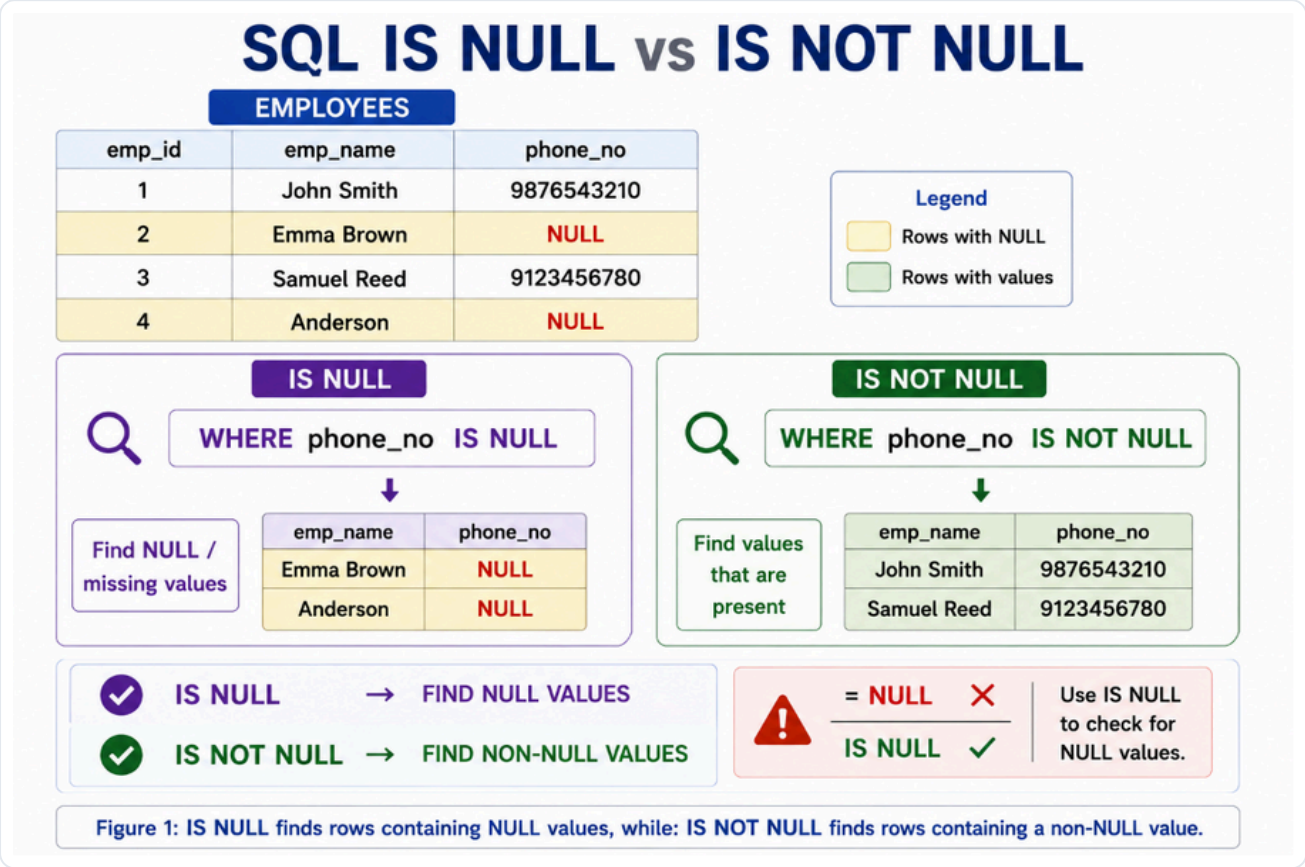


Figure 1: IS NULL finds rows containing NULL values, while IS NOT NULL finds rows containing a non-NULL value.

Practice Table

This topic uses an `employees` table with optional contact information:

emp_id	emp_name	phone_no	salary
1	John Smith	9876543210	45000
2	Emma Brown	`NULL`	60000
3	Samuel Reed	9123456780	32000
4	Anderson	`NULL`	55000

IS NULL

`IS NULL` finds rows where a specific column has no value recorded.

```sql
SELECT *
FROM employees
WHERE phone_no IS NULL;
```

This returns employees whose phone number is missing:

```text
emp_id | emp_name   | phone_no | salary
2      | Emma Brown | NULL     | 60000
4      | Anderson   | NULL     | 55000
```

> *IS NULL → Filter for empty/missing entries*

IS NOT NULL

`IS NOT NULL` filters for rows that have an actual value stored.

```sql
SELECT *
FROM employees
WHERE phone_no IS NOT NULL;
```

This returns employees who have a recorded phone number:

```
TEXT
Copy

emp_id | emp_name    | phone_no   | salary
1      | John Smith  | 9876543210 | 45000
3      | Samuel Reed | 9123456780 | 32000
```

> *IS NOT NULL → Filter for present entries*

Why `= NULL` Fails (Three-Valued Logic)

In SQL, comparing anything to `NULL` using `= NULL` yields **UNKNOWN**, never `TRUE`:

```sql
SQL
Copy

-- ✗ INCORRECT: Returns 0 rows every time!
SELECT * FROM employees WHERE phone_no = NULL;

-- ✓ CORRECT:
SELECT * FROM employees WHERE phone_no IS NULL;
```

SQL operates on **Three-Valued Logic**: 1. `TRUE` 2. `FALSE` 3. `UNKNOWN`

Since `NULL` means "unknown", the database does not know whether one unknown equals another unknown. Therefore, `NULL = NULL` is `UNKNOWN`, and rows with `UNKNOWN` conditions are excluded from `WHERE` clauses.

```
TEXT
Copy

      = NULL   ──► ✗ FAILS (Evaluates to UNKNOWN)
   IS NULL   ──► ✓ WORKS (Correct SQL Operator)
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Checking missing data with `WHERE col = NULL`	Always use `WHERE col IS NULL`
Checking present data with `WHERE col != NULL`	Always use `WHERE col IS NOT NULL`
Assuming `NULL` is equivalent to `0` or empty string `''`	`0` is a number; `''` is a string; `NULL` is the absence of any value
Expecting `COUNT(column)` to count NULLs	`COUNT(column)` ignores NULLs; only `COUNT(*)` counts all rows

Placement Quick Points

```TEXT
IS NULL
→ FINDS UNRECORDED / MISSING VALUES

IS NOT NULL
→ FINDS RECORDED / VALID VALUES

= NULL
→ ALWAYS INVALID IN SQL
```

- `NULL` represents missing, unknown, or not-applicable information.
- Never use `= NULL` or `!= NULL`; always use `IS NULL` and `IS NOT NULL`.
- `NULL` is not zero, not blank space, and not empty string.
- In three-valued logic, comparisons with NULL evaluate to `UNKNOWN`.

Interview Questions

Q1. What does NULL represent in SQL?

`NULL` signifies missing, unrecorded, or unknown data. It is a marker indicating the absence of a value.

Q2. Why can't we use `= NULL` to test for NULL values?

Because any equality comparison with `NULL` evaluates to `UNKNOWN` in SQL's three-valued logic, which evaluates as false for `WHERE` clause row inclusion.

Q3. What is the difference between NULL, 0, and an empty string `''`?

- `0` is a defined numeric value.
- `''` is a defined string of zero length.
- `NULL` is completely undefined / no value assigned.

Q4. How does `COUNT(column)` treat NULL values?

`COUNT(column_name)` excludes NULL values, counting only non-NULL occurrences. `COUNT(*)` counts total rows regardless of NULLs.

Q5. How do aggregate functions like SUM and AVG handle NULLs?

They silently ignore NULL values when calculating aggregate results.

Practice & Hands-On Exercises

Using the `employees` table:

1. Retrieve all employees who do not have a phone number:

```sql
SELECT emp_name FROM employees WHERE phone_no IS NULL;
```

1. Retrieve all employees who have a verified phone number:

```sql
SELECT emp_name, phone_no FROM employees WHERE phone_no IS NOT NULL;
```

1. Test what happens when you run `SELECT * FROM employees WHERE phone_no = NULL;`.
2. Explain why `NULL = NULL` does not evaluate to `TRUE`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
IS NULL     → FIND MISSING VALUES
IS NOT NULL → FIND NON-NULL VALUES

= NULL ✗    → NEVER USE EQUALITY WITH NULL
```

> ***Use IS NULL to check for missing values and IS NOT NULL to find rows with values present.***

End of IS NULL / IS NOT NULL

8.7 LIKE / NOT LIKE and Wildcards

Level: 8 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`LIKE` is used to **match text against a pattern**.

`NOT LIKE` is used to **exclude text that matches a pattern**.

Operator	Purpose
`LIKE`	Matches a pattern
`NOT LIKE`	Excludes a pattern

Two main wildcards are used:

Wildcard	Meaning
`%`	Any number of characters, including zero
`_`	Exactly one character

> ***LIKE → Match a pattern***
> ***NOT LIKE → Exclude a pattern***

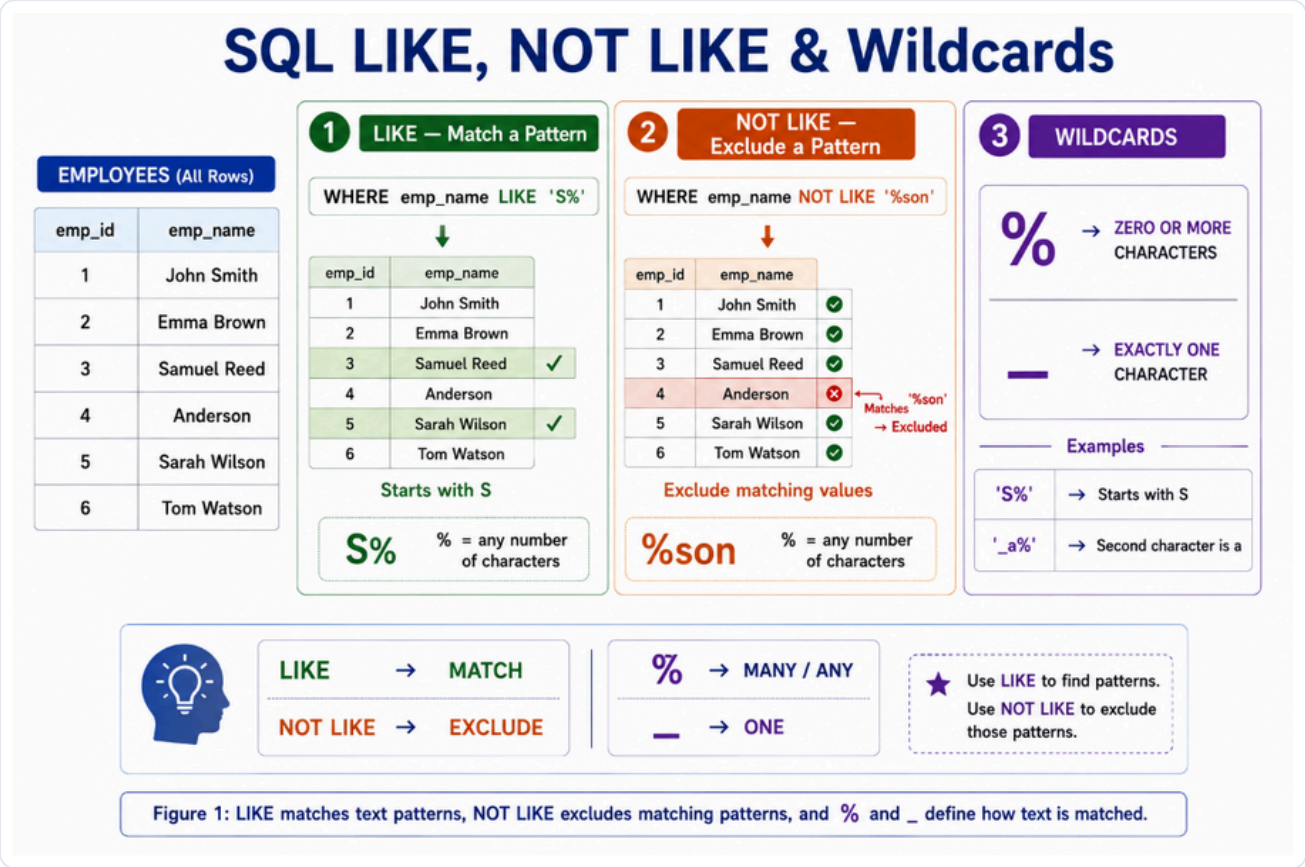


Figure 1: LIKE matches text patterns, NOT LIKE excludes matching patterns, and % and _ define how text is matched.

Practice Table

This topic reuses the `employees` table from **8.1 The WHERE Clause**.

emp_id	emp_name	dept_id	salary
1	John Smith	10	45000
2	Emma Brown	20	60000
3	Samuel Reed	10	32000
4	Anderson	30	55000
5	Sarah Wilson	20	48000
6	Tom Watson	10	51000

LIKE

`LIKE` matches text using a pattern.

Starts With

```sql
SELECT *
FROM employees
WHERE emp_name LIKE 'S%';
```

`S%` means:

> *Starts with `S`, followed by any number of characters.*

This matches:

```
TEXT
Copy

Samuel Reed
Sarah Wilson
```

Ends With

```
SQL
Copy

SELECT *
FROM employees
WHERE emp_name LIKE '%son';
```

`%son` means:

> *Ends with* `son`.

For example:

```
TEXT
Copy

Anderson
```

Contains

```
SQL
Copy

SELECT *
FROM employees
WHERE emp_name LIKE '%am%';
```

`%am%` means the text `am` can appear anywhere in the value.

> *LIKE → Find text that matches a pattern*

NOT LIKE

`NOT LIKE` excludes values that match the pattern.

```sql
SELECT *
FROM employees
WHERE emp_name NOT LIKE '%son';
```

This returns names that do **not** end with `son`.

> *NOT LIKE → Exclude matching patterns*

Wildcard `%`

`%` matches **zero or more characters**.

Examples:

```text
'a%'    → Starts with a
'%a'    → Ends with a
'%a%'   → Contains a
```

For example:

```sql
SELECT *
FROM employees
WHERE emp_name LIKE 'J%';
```

matches names beginning with `J`.

Wildcard `_`

`_` matches **exactly one character**.

Example:

```sql
SELECT *
FROM employees
WHERE emp_name LIKE '_a%';
```

Here:

```text
_ → first character
a → second character
% → remaining characters
```

So the second character must be `a`.

Common Patterns

Pattern	Meaning
`'S%'`	Starts with `S`
`'%son'`	Ends with `son`
`'%am%'`	Contains `am`
`'S_'`	Starts with `S` and has exactly 2 characters
`'_a%'`	Second character is `a`
`'__a%'`	Third character is `a`

Easy Memory Trick

```text
% → MANY / ANY CHARACTERS
_ → ONE CHARACTER
```

LIKE vs NOT LIKE

```text
LIKE
→ MATCH THE PATTERN

NOT LIKE
→ EXCLUDE THE PATTERN
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`%` means exactly one character	`%` means zero or more characters
`_` means any number of characters	`_` means exactly one character
`LIKE 'S%'` finds names containing S anywhere	It finds values starting with `S`
`LIKE '%son'` finds values containing `son` anywhere	It finds values ending with `son`
`NOT LIKE` removes data	It only filters the query result
`=` and `LIKE` always behave the same	`LIKE` is used for pattern matching

Placement Quick Points

```text
LIKE
→ MATCH PATTERN

NOT LIKE
→ EXCLUDE PATTERN

%
→ ZERO OR MORE CHARACTERS

_
→ EXACTLY ONE CHARACTER
```

- `LIKE` is used for pattern matching.

- `NOT LIKE` excludes matching patterns.
- `%` matches zero or more characters.
- `_` matches exactly one character.
- `LIKE 'S%'` finds values starting with `S`.
- `LIKE '%son'` finds values ending with `son`.
- `LIKE '%am%'` finds values containing `am`.

Interview Questions

Q1. What is LIKE in SQL?

`LIKE` is used to match text against a specified pattern.

Q2. What is NOT LIKE?

`NOT LIKE` returns rows whose values do not match the specified pattern.

Q3. What does `%` mean?

`%` matches zero or more characters.

Q4. What does `_` mean?

`_` matches exactly one character.

Q5. What does `LIKE 'S%'` mean?

It matches values that start with `S`.

Q6. What does `LIKE '%son'` mean?

It matches values that end with `son`.

Q7. What does `LIKE '_a%'` mean?

The second character must be `a`, with any number of characters allowed after it.

Practice & Hands-On Exercises

Using the `employees` table:

1. Find employees whose name starts with `S`:

```sql
SELECT * FROM employees WHERE emp_name LIKE 'S%';
```

1. Find employees whose name ends with `son`:

```sql
SELECT * FROM employees WHERE emp_name LIKE '%son';
```

1. Find employees whose name contains `am`:

```sql
SELECT * FROM employees WHERE emp_name LIKE '%am%';
```

1. Find employees whose name does not end with `son`:

```sql
SELECT * FROM employees WHERE emp_name NOT LIKE '%son';
```

1. Find names where the second character is `a`:

```sql
SELECT * FROM employees WHERE emp_name LIKE '_a%';
```

1. Explain the difference between `%` and `_`.
2. Explain the difference between `LIKE` and `NOT LIKE`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```TEXT
LIKE     → MATCH
NOT LIKE → EXCLUDE

% → ANY NUMBER OF CHARACTERS
_ → EXACTLY ONE CHARACTER
```

Copy

> ***LIKE and NOT LIKE are used for text pattern matching, while `%` and `_` define how the pattern should match.***

End of LIKE / NOT LIKE and Wildcards

9.1 What is a Data Model? (Conceptual / Logical / Physical)

Level: 9 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **data model** is a way to represent an organization's data and the **relationships between that data**.

It helps us move from a business requirement to an actual database design.

Data modeling is commonly explained in three levels:

```text
CONCEPTUAL
    ↓
LOGICAL
    ↓
PHYSICAL
```

Copy

> *Conceptual → What the business needs*
> *Logical → How the data is organized*
> *Physical → How it is implemented in a database*

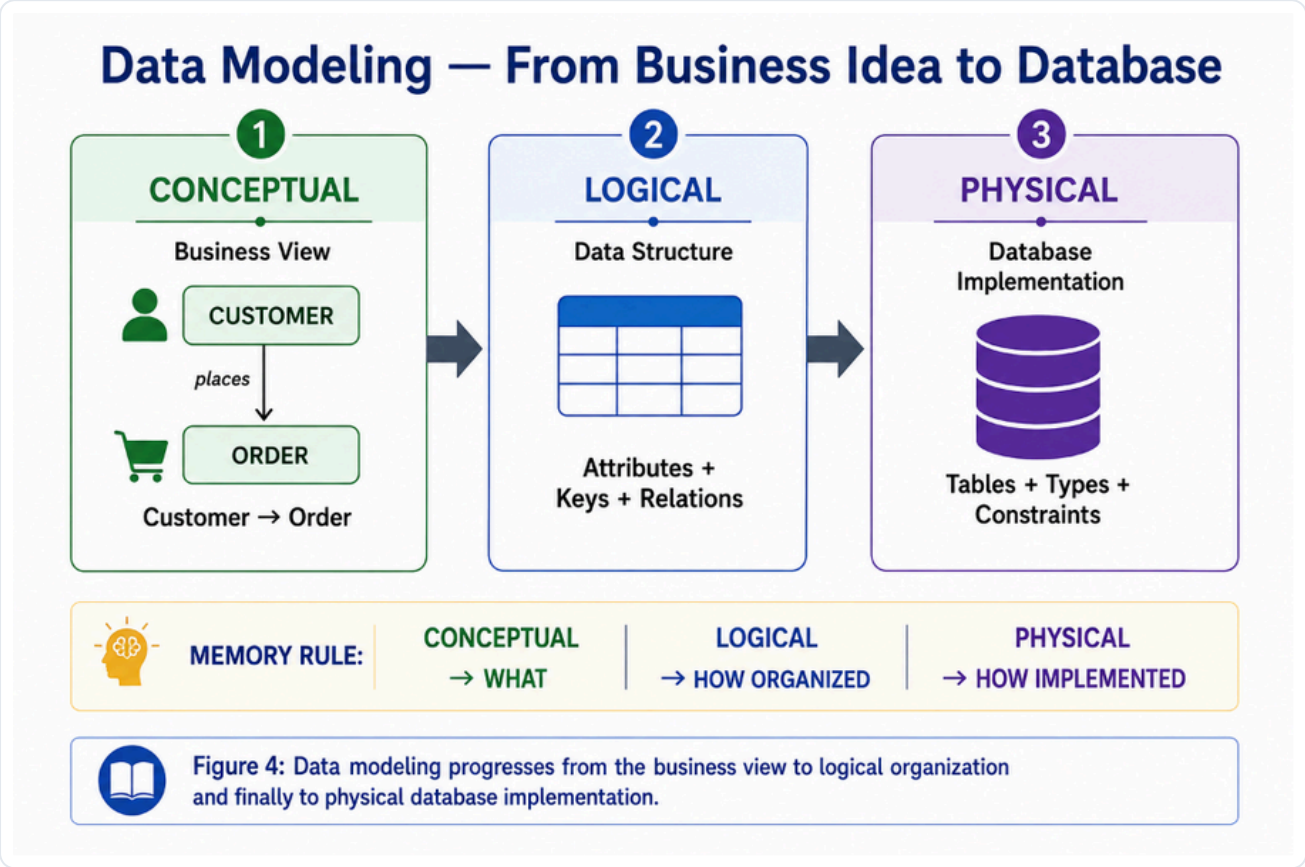


Figure 4: Data modeling progresses from the business view to logical organization and finally to physical database implementation.

Conceptual Data Model

The **Conceptual Data Model** is the highest-level view.

It focuses on:

- Main entities
- Relationships between entities
- Business requirements

It does not usually include database-specific details.

Example:

```text
[ Customer ]
      |
      | places
      ↓
[ Order ]
```

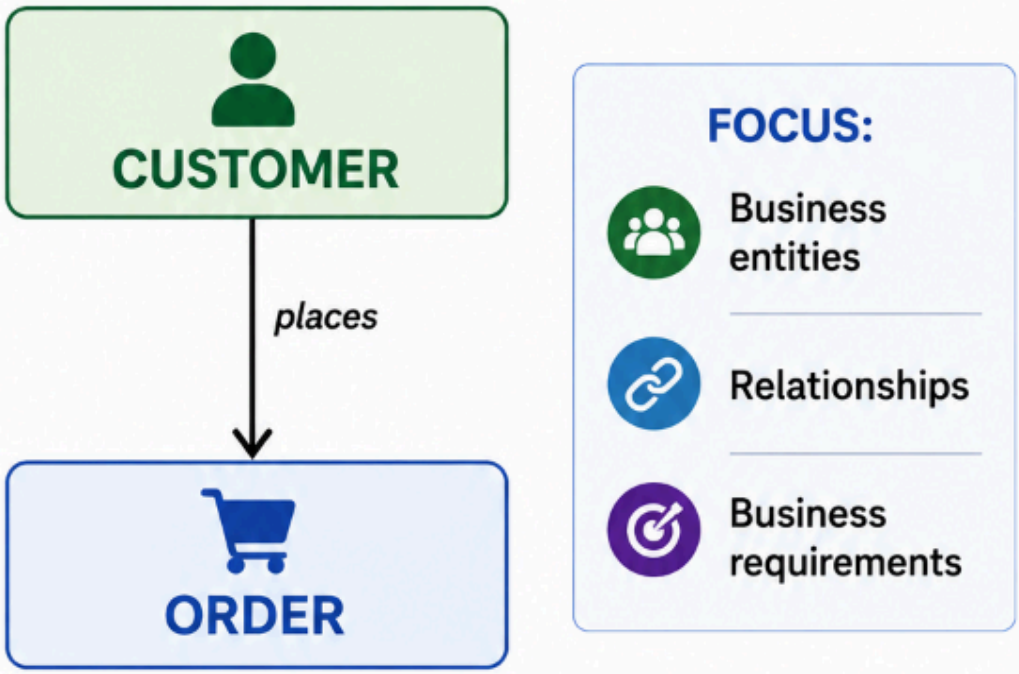
At this stage, we are mainly asking:

> *"What things exist in the business, and how are they related?"*

Conceptual Data Model — Business View

HIGH-LEVEL BUSINESS VIEW

A conceptual data model shows what business entities exist and how they are related.



RELATIONSHIP:

CUSTOMER → *places* → ORDER

QUESTION:

"What things exist in the business and how are they related?"

Figure 1: A conceptual data model provides a high-level business view of entities and their relationships.

Figure 1: A conceptual data model provides a high-level business view of entities and their relationships.

Logical Data Model

The **Logical Data Model** adds more detail.

It defines:

- Attributes
- Keys
- Relationships
- Data organization

Example:

```text
Customer(
    customer_id,
    name,
    email
)

Order(
    order_id,
    order_date,
    customer_id
)
```

Copy

At this stage, we are asking:

> *"What data do we need, and how should it be logically organized?"*

The logical model is generally independent of a specific DBMS.

Logical Data Model — Detailed Data Structure

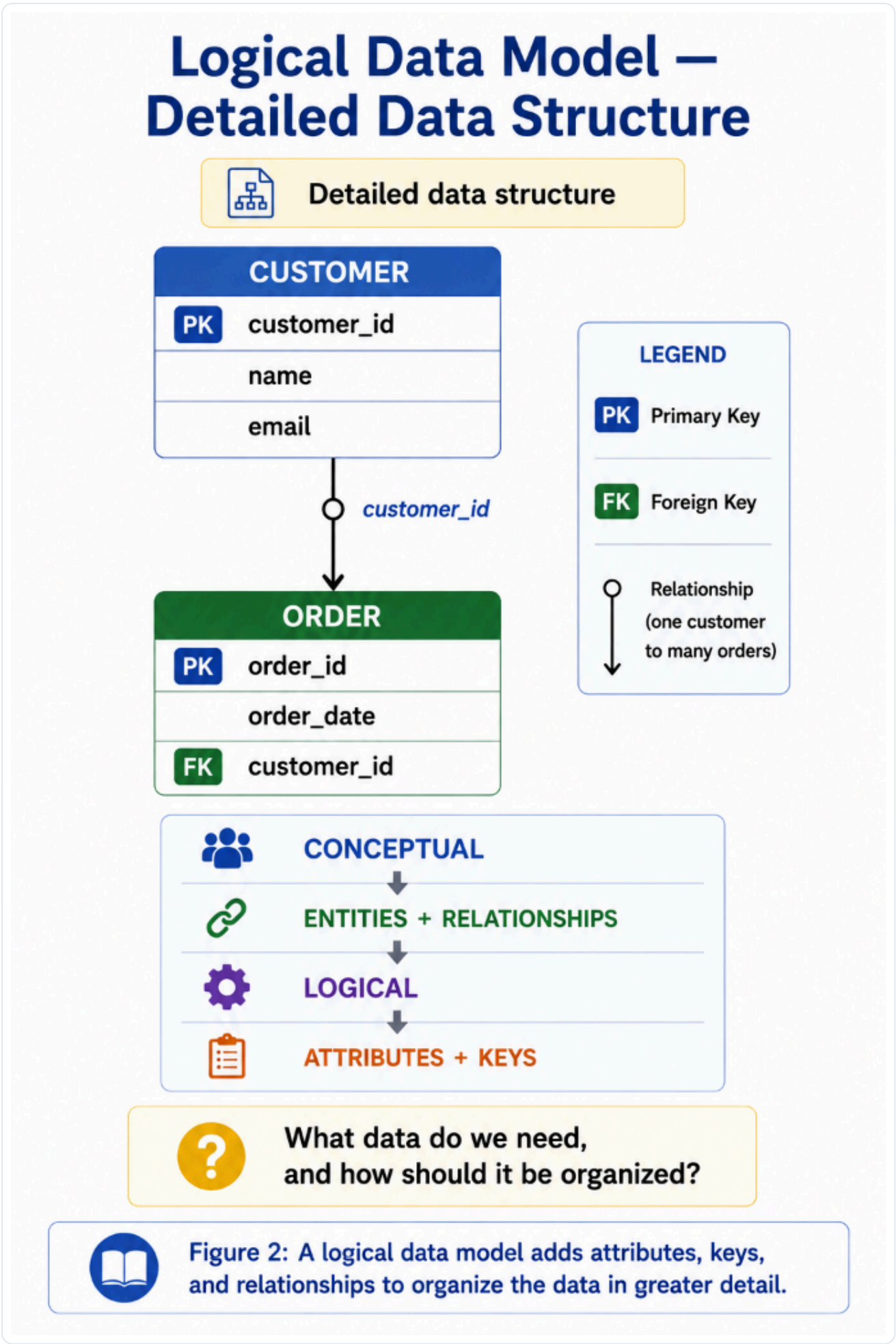


Figure 2: A logical data model adds attributes, keys, and relationships to organize the data in greater detail.

Figure 2: A logical data model adds attributes, keys, and relationships to organize the data in greater detail.

Physical Data Model

The **Physical Data Model** describes how the design will actually be implemented in a particular database system.

It can include:

- Table names
- Column names
- Data types
- Constraints
- Indexes
- DBMS-specific features

For example, in PostgreSQL:

```sql
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);
```

Here we are deciding exactly how the logical design will be implemented.

> *Physical model → Actual database implementation*

Physical Data Model — Actual Database Implementation

Actual database implementation

LOGICAL MODEL (Simplified)

CUSTOMER

customer_id

name

email

Implement in PostgreSQL

PHYSICAL MODEL (PostgreSQL Table)

customers

Column Name	Data Type
customer_id	INT
name	VARCHAR(100)
email	VARCHAR(100)

PRIMARY KEY (customer_id)

UNIQUE (email)

PostgreSQL

LOGICAL DESIGN

↓

TABLES

↓

COLUMNS + DATA TYPES

↓

CONSTRAINTS

↓

PHYSICAL DATABASE

The physical model describes how the logical design is actually implemented in a specific DBMS.

Figure 3: A physical data model shows how the logical design is implemented using database-specific tables, data types, and constraints.

Figure 3: A physical data model shows how the logical design is implemented using database-specific tables, data types, and constraints.

Simple Comparison

Level	Focus	Example
Conceptual	Business entities and relationships	Customer → Order
Logical	Attributes, keys, relationships	`Customer(customer_id, name, email)`
Physical	Tables, data types, indexes, constraints	PostgreSQL `CREATE TABLE`

Easy Memory Trick

```text
CONCEPTUAL → BUSINESS IDEA
LOGICAL    → DATA DESIGN
PHYSICAL   → DATABASE IMPLEMENTATION
```

One Example: Online Shopping

Suppose we are designing an online shopping system.

Conceptual

```text
Customer → places → Order
```

Logical

```text
Customer(
    customer_id,
    name
)

Order(
    order_id,
    order_date,
    customer_id
)
```

Physical

```sql
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT REFERENCES customers(customer_id)
);
```

The same business requirement is gradually converted into a database implementation.

Why Data Modeling Is Important

Data modeling helps teams:

- Understand business requirements
- Organize data properly
- Define relationships
- Reduce design problems
- Create a clear database structure

The three levels provide different views of the same system.

```text
BUSINESS REQUIREMENT
        ↓
CONCEPTUAL MODEL
        ↓
LOGICAL MODEL
        ↓
PHYSICAL MODEL
        ↓
DATABASE
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Conceptual and physical models are the same	Conceptual is high-level; physical is implementation-specific
Logical model contains SQL code	SQL is usually part of the physical implementation
Physical model is only about storage disks	It also includes tables, columns, data types, constraints, indexes, and DBMS-specific details
Every model must contain the same level of detail	Each level has a different purpose
Data modeling is only for database administrators	Developers, data engineers, architects, and business teams can all use data models

Placement Quick Points

```text
CONCEPTUAL
→ ENTITIES + RELATIONSHIPS

LOGICAL
→ ATTRIBUTES + KEYS + RELATIONSHIPS

PHYSICAL
→ TABLES + TYPES + CONSTRAINTS + INDEXES
```

- A data model represents data and its relationships.
- **Conceptual** models focus on business concepts.

- **Logical** models add attributes, keys, and relationships.
- **Physical** models describe the actual database implementation.
- Data modeling moves from business requirements toward a concrete database design.

Interview Questions

Q1. What is a data model?

A data model is a representation of data, its structure, and its relationships.

Q2. What are the three levels of data modeling?

```
text Conceptual Logical Physical
```

Q3. What is a conceptual data model?

It is a high-level representation of business entities and their relationships without database-specific details.

Q4. What is a logical data model?

It defines attributes, keys, and relationships in greater detail while generally remaining independent of a specific DBMS.

Q5. What is a physical data model?

It describes how the logical design is implemented in a specific database system using tables, columns, data types, constraints, indexes, and other details.

Q6. What is the main difference between logical and physical models?

```
text Logical → What the data structure should be Physical → How it is implemented in a DBMS
```

Practice & Hands-On Exercises

1. Explain conceptual, logical, and physical data models in your own words.
2. Convert `Customer → Order` into a logical model.
3. Identify the attributes in the logical model.
4. Identify the primary and foreign keys.
5. Write PostgreSQL tables for the logical design.

💡 *Tip: Test your schema queries using the in-browser interactive runner above.*

Key Takeaway

```
CONCEPTUAL
→ BUSINESS VIEW

LOGICAL
→ DATA STRUCTURE

PHYSICAL
→ ACTUAL DATABASE
```

> ***Data modeling moves from a high-level business idea to a detailed logical design and finally to a physical database implementation.***

End of What is a Data Model? (Conceptual / Logical / Physical)

9.2 ER Diagrams — Entities, Attributes, Relationships

Level: 9 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

An **ER Diagram (Entity-Relationship Diagram)** is a visual way to design and understand a database.

It shows:

- **Entities** → What things are stored
- **Attributes** → Properties of those things
- **Relationships** → How those things are connected

For example:

```text
ENTITY
  ↓
ATTRIBUTES
  ↓
RELATIONSHIP
```

> *ER Diagram → Visual blueprint of a database.*

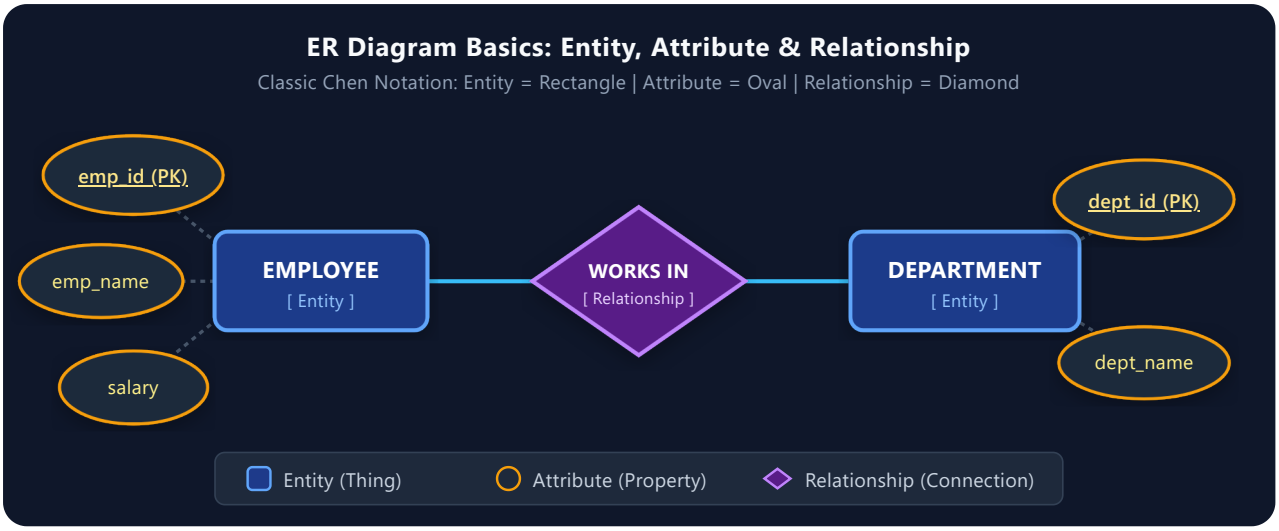


Figure 1: An ER diagram represents entities (rectangles), their attributes (ovals), and the relationships between entities (diamonds).

Entities

An **entity** represents a real-world object or concept that we want to store data about.

Examples:

```text
Student
Employee
Customer
Product
Department
Order
```

For example, in a college database:

```text
Student
Course
Department
```

Can all be entities.

Attributes

An **attribute** describes a property of an entity.

For example, an `Employee` may have:

```
TEXT

Employee
├── emp_id
├── emp_name
├── department
└── salary
```

Here, `emp_id`, `emp_name`, `department`, and `salary` are attributes.

> ***Entity = Thing***
> ***Attribute = Property of the thing***

Relationships

A **relationship** describes how two or more entities are connected.

For example:

This tells us that an employee belongs to a department.

Common relationship examples:

```text
Customer → places → Order
Student  → enrolls in → Course
Employee → works in → Department
```

> *Relationship = Connection between entities*

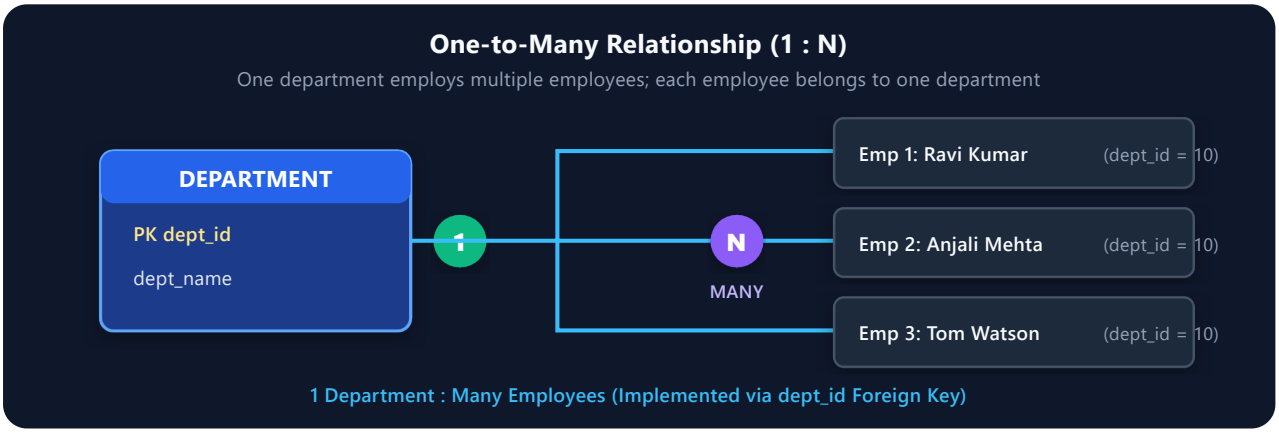


Figure 2: A one-to-many relationship allows one department to be associated with many employees.

ER Diagram and SQL Tables

An ER diagram is often created **before implementing the database tables**.

For example:

```
TEXT

ER DESIGN
   ↓
TABLE DESIGN
   ↓
SQL IMPLEMENTATION
```

The relationship:

```
TEXT

Department
     ↓
   Employee
```

Can later be implemented using a foreign key.

```sql
SQL

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT REFERENCES departments(dept_id)
);
```

Here, `dept_id` connects the employee to the department.

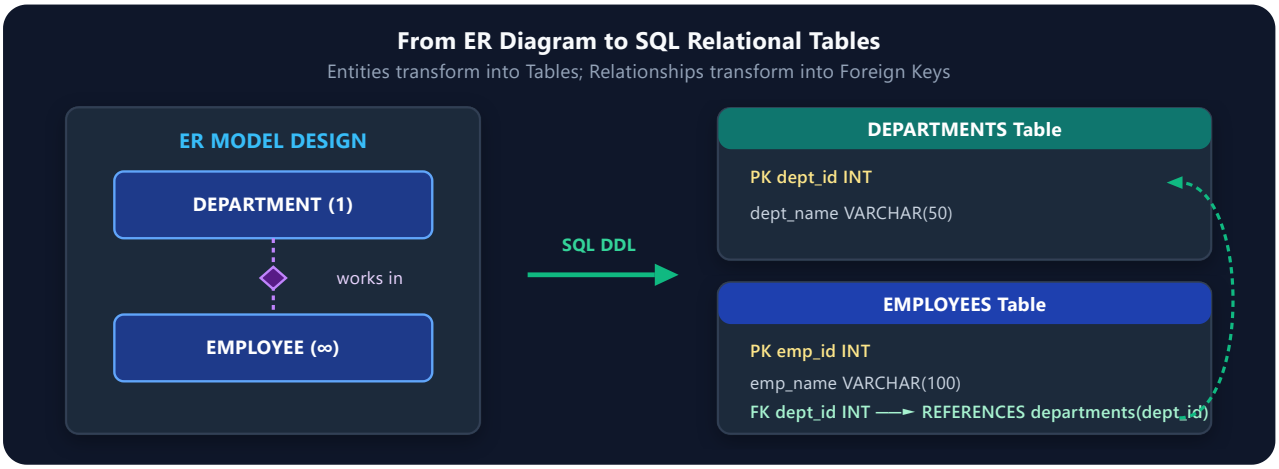


Figure 3: An ER relationship can be implemented in SQL tables using keys such as a foreign key.

Simple ER Example

Consider an online shopping system.

Entities:

```
TEXT

Customer
Product
Order
```

Copy

Relationships:

```
TEXT

Customer → places → Order
Order → contains → Product
```

Copy

A simplified ER design might look like:

```
TEXT

Customer
   ↓
 places
   ↓
Order
   ↓
contains
   ↓
Product
```

Copy

This gives the database designer a clear picture before creating the actual tables.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Entity and attribute are the same	Entity is the object; attribute describes it
A relationship is a table	A relationship describes how entities are connected
ER diagrams are only used after tables are created	They are often used during database design before implementation
Every entity must have only three attributes	An entity can have as many appropriate attributes as the design requires
A foreign key is the same as a relationship	The foreign key is a database implementation mechanism for representing certain relationships

Placement Quick Points

```text
ENTITY
→ REAL-WORLD OBJECT

ATTRIBUTE
→ PROPERTY OF ENTITY

RELATIONSHIP
→ CONNECTION BETWEEN ENTITIES
```

- **Entity** represents a real-world object or concept.
- **Attribute** describes an entity.

- **Relationship** connects entities.
- ER diagrams help visualize database structure before implementation.
- Relationships can be implemented using keys such as foreign keys.
- One-to-many is a common relationship pattern.

Interview Questions

Q1. What is an ER diagram?

An ER diagram is a visual representation of entities, attributes, and relationships used to design a database.

Q2. What is an entity?

An entity is a real-world object or concept about which data is stored.

Q3. What is an attribute?

An attribute is a property that describes an entity.

Q4. What is a relationship?

A relationship describes how two or more entities are connected.

Q5. Give an example of a one-to-many relationship.

One department can have many employees.

```
text Department 1 ↓ Employees Many
```

Q6. How is an ER relationship implemented in SQL?

It can be represented using keys such as a primary key and foreign key.

Q7. Why are ER diagrams useful?

> They help developers and database designers understand the structure and relationships of the data before implementing the database.

Practice & Hands-On Exercises

1. Identify the entities in a college database.
2. Give three attributes for a `Student` entity.
3. Give an example of a relationship between `Student` and `Course`.
4. Draw an ER diagram for `Department` and `Employee`.
5. Identify the one-to-many relationship in your diagram.
6. Explain how that relationship could be implemented using a foreign key:

```text
TEXT
```
Copy

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    dept_id INT REFERENCES departments(dept_id)
);
```

💡 *Tip: Test your queries and schema definitions using the in-browser interactive runner above.*

Key Takeaway

```sql
ENTITY
→ WHAT

ATTRIBUTE
→ DETAILS

RELATIONSHIP
→ HOW THEY CONNECT
```

> *ER diagrams provide a visual blueprint of a database by showing entities, their attributes, and the relationships between them.*

End of ER Diagrams — Entities, Attributes, Relationships

9.3 Crow's Foot Notation

Level: 9 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Crow's Foot Notation is a visual notation used in ER diagrams to show **how many records can be related** between entities.

The main symbols represent:

Symbol	Meaning
`|`	One
`0`	Zero / Optional
`<`	Many

These symbols can be combined:

```
TEXT

||  → Exactly one
0|  → Zero or one
|<  → One or many
0<  → Zero or many
```

> *Crow's Foot notation → Shows relationship cardinality.*

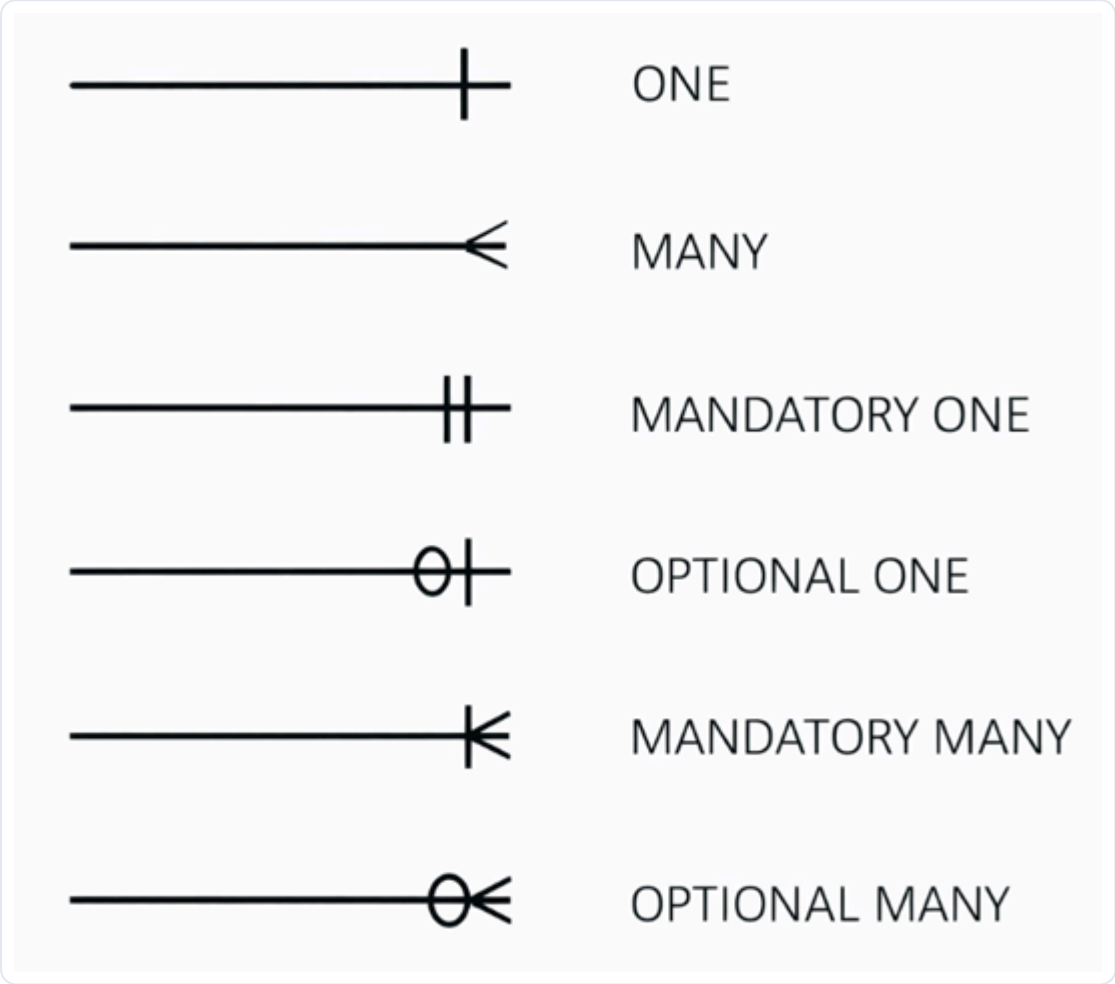


Figure 1: Crow's Foot symbols show whether a relationship is mandatory, optional, one, or many.

Example

Consider:

```
TEXT
DEPARTMENT |—< EMPLOYEE
```

Copy

This means:

> *One department can have many employees.*

The symbol near `DEPARTMENT` represents **one**, while the crow's foot near `EMPLOYEE` represents **many**.

Another example:

```
TEXT
Copy

CUSTOMER ||—<O ORDER
```

This represents an optional-many relationship on the `ORDER` side: a customer may have zero or many orders, depending on the business rules.

Quick Memory

```
TEXT
Copy

|  → ONE
O  → OPTIONAL / ZERO
<  → MANY
```

> *Bar = One, Circle = Zero, Crow's Foot = Many*

Placement Quick Points

```TEXT
|   → ONE
O   → OPTIONAL / ZERO
<   → MANY

||  → MANDATORY ONE
O|  → OPTIONAL ONE
|<  → MANDATORY MANY
O<  → OPTIONAL MANY
```

- Crow's Foot notation represents **cardinality and optionality** in ER diagrams.
- `|` means one.
- `O` means zero/optional.
- `<` means many.
- Symbols can be combined to show relationships such as zero-or-one or zero-or-many.

Interview Questions

Q1. What is Crow's Foot notation?

> It is a visual notation used in ER diagrams to represent relationship cardinality and optionality.

Q2. What does the crow's foot mean?

> It represents **many**.

Q3. What does a circle mean?

> It represents **zero or optional** participation.

Q4. What does a bar mean?

> It represents **one**.

Q5. What does `0<` mean?

> It means **zero or many** (optional many).

Practice & Hands-On Exercises

1. What does `|` represent?
2. What does `0` represent?
3. What does `<` represent?
4. Interpret `|<`.
5. Interpret `0<`.
6. Draw a Crow's Foot relationship where **one department has many employees**.

💡 *Tip: Test your queries and schema definitions using the in-browser interactive runner above.*

Key Takeaway

```text
|  → ONE
0  → ZERO / OPTIONAL
<  → MANY
```

> *Crow's Foot notation makes ER relationships easy to read by showing one, zero, and many visually.*

End of Crow's Foot Notation

9.4 Cardinality (1:1, 1:N, M:N)

Level: 9 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Cardinality describes how many records in one table can be related to records in another table.

The three common types are:

Type	Meaning	Example
1:1	One ↔ One	Person ↔ Passport
1:N	One ↔ Many	Department ↔ Employees
M:N	Many ↔ Many	Students ↔ Courses

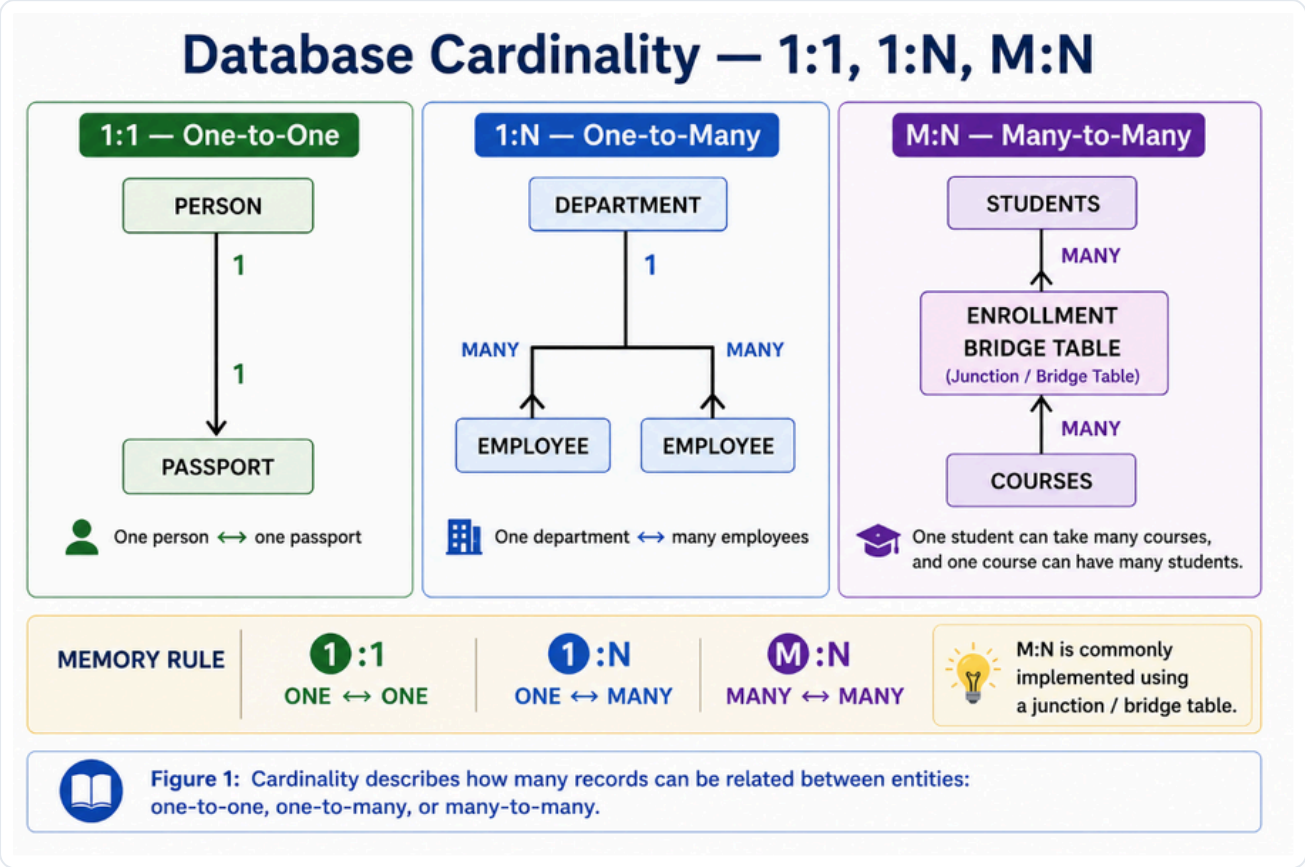


Figure 1: Cardinality describes how many records can be related between entities: one-to-one, one-to-many, or many-to-many.

1:1 — One-to-One

One record in Table A is related to one record in Table B.

Example:

```
TEXT
PERSON 1 ——— 1 PASSPORT
```

A person has one passport, and that passport belongs to one person.

> *1:1 → One to One*

1:N — One-to-Many

One record in Table A can relate to many records in Table B.

Example:

```
TEXT
DEPARTMENT 1 ——< EMPLOYEES
```

Copy

One department can have many employees.

> *1:N → One to Many*

M:N — Many-to-Many

Many records in Table A can relate to many records in Table B.

Example:

```
TEXT
STUDENTS >——< COURSES
```

Copy

A student can take many courses, and a course can have many students.

```text
Students
   ↓
Enrollment
   ↓
Courses
```

M:N Example in SQL

```sql
CREATE TABLE enrollment (
    student_id INT REFERENCES students(student_id),
    course_id INT REFERENCES courses(course_id)
);
```

The `enrollment` table connects students and courses.

A query can retrieve the relationship:

```sql
SELECT s.student_name, c.course_name
FROM students s
JOIN enrollment e
    ON s.student_id = e.student_id
JOIN courses c
    ON e.course_id = c.course_id;
```

Quick Comparison

> *M:N relationships are usually implemented using a junction table.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
1:N means one table has many rows	It describes the relationship between records/entities
M:N is directly stored in one foreign-key column	It is normally implemented using a junction table
Cardinality and foreign key are the same	Cardinality describes the relationship; a foreign key helps implement it
One student can have only one course in an M:N relationship	A student can have many courses

Placement Quick Points

- **1:1** → One record relates to one record.
- **1:N** → One record relates to many records.
- **M:N** → Many records relate to many records.
- M:N relationships are commonly implemented using a junction table.
- Cardinality helps define database relationships.

Interview Questions

Q1. What is cardinality?

> Cardinality describes how many records can participate in a relationship between entities.

Q2. What is 1:1?

> One record relates to exactly one record.

Q3. What is 1:N?

> One record can relate to many records.

Q4. What is M:N?

> Many records can relate to many records.

Q5. How is M:N implemented in a relational database?

> Using a **junction/bridge table** containing foreign keys referencing the two related tables.

Practice & Hands-On Exercises

1. Give one real-world example of `1:1`.
2. Give one example of `1:N`.
3. Give one example of `M:N`.
4. Explain why a junction table is needed for M:N.
5. Identify the cardinality between `Department` and `Employee`.

💡 *Tip: Test your queries and schema definitions using the in-browser interactive runner above.*

Key Takeaway

```text
1:1 → ONE TO ONE
1:N → ONE TO MANY
M:N → MANY TO MANY
```

Cardinality describes how many records can be related, with M:N relationships commonly implemented through a junction table.

End of Cardinality (1:1, 1:N, M:N)

9.5 Mandatory vs Optional Relationships

Level: 9 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A relationship can be **mandatory** or **optional** depending on whether the related record is required.

Type	Meaning
Mandatory	Relationship is required
Optional	Relationship is not required

Example

Mandatory:

```sql
customer_id INT NOT NULL REFERENCES customers(customer_id)
```

Every order must have a customer.

Optional:

```sql
manager_id INT REFERENCES employees(emp_id)
```

`manager_id` can be `NULL`, so an employee may have no manager.

NOT NULL → Mandatory
NULL allowed → Optional

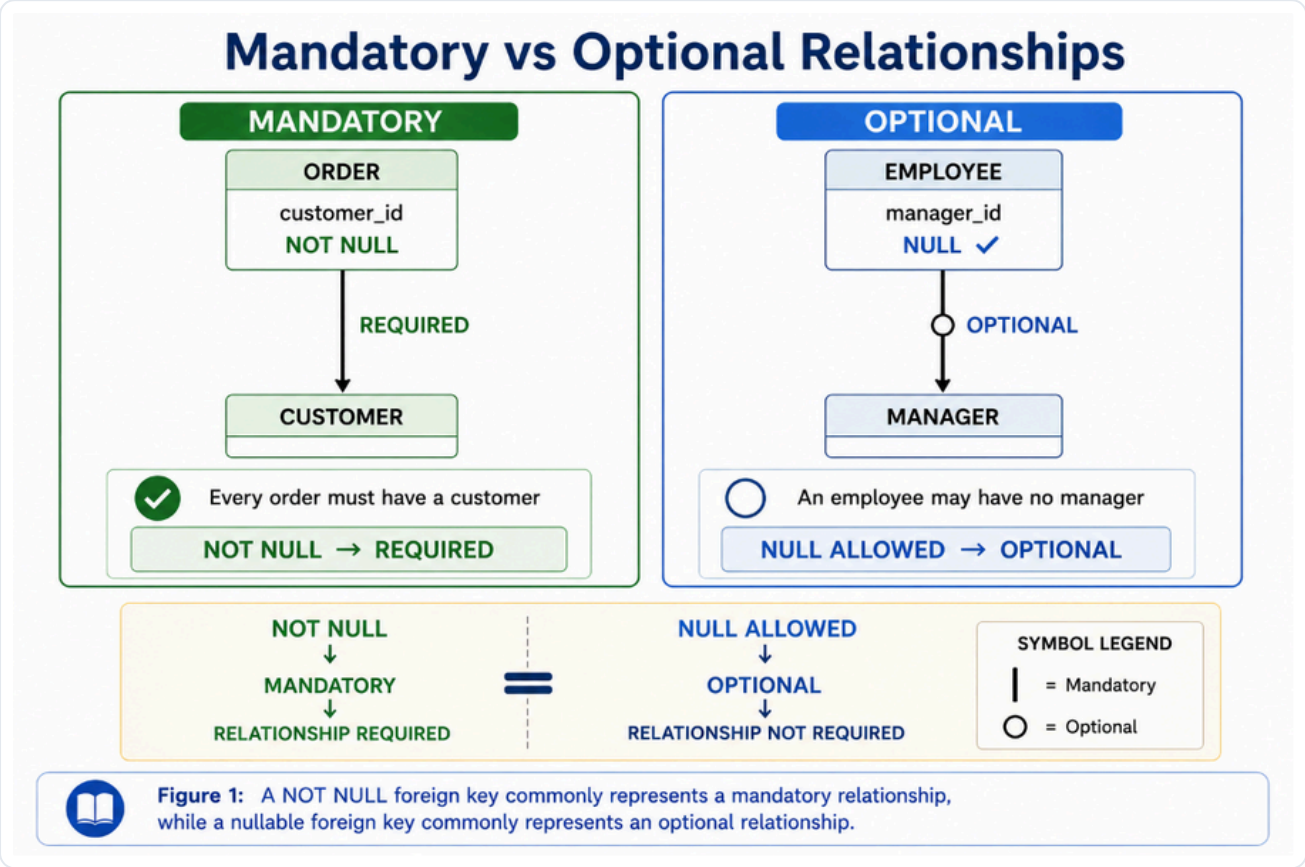


Figure 1: A NOT NULL foreign key commonly represents a mandatory relationship, while a nullable foreign key commonly represents an optional relationship.

Quick Points

```text
MANDATORY → NOT NULL → REQUIRED
OPTIONAL  → NULL     → NOT REQUIRED
```

- Mandatory means the relationship must exist.
- Optional means the relationship may not exist.

- A nullable foreign key commonly represents an optional relationship.
- A `NOT NULL` foreign key commonly represents a mandatory relationship.

Interview Questions

Q1. What is a mandatory relationship?

> A relationship where the related record is required.

Q2. What is an optional relationship?

> A relationship where the related record is not required.

Q3. How is a mandatory relationship commonly implemented?

> Using a foreign key with `NOT NULL`.

Q4. How is an optional relationship commonly implemented?

> Using a foreign key that allows `NULL`.

Practice

1. Identify whether `customer_id NOT NULL` is mandatory or optional.
2. Identify whether a nullable `manager_id` is mandatory or optional.
3. Give one real-world example of each.

💡 *Tip: Test your queries and schema definitions using the in-browser interactive runner above.*

Key Takeaway

```text
NOT NULL → MANDATORY
NULL     → OPTIONAL
```

> *Mandatory relationships require a related record; optional relationships allow the relationship to be absent.*

End of Mandatory vs Optional Relationships

10.1 What are Joins? (Overview)

Level: 10 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **JOIN** is used to combine related data from two or more tables.

For example:

```text
Students
   +
Courses
   ↓
JOIN
   ↓
Combined Result
```

Joins are useful when related information is stored in separate tables.

There are six common join types:

Join Type	Main Idea
INNER JOIN	Matching rows from both tables
LEFT JOIN	All left rows + matching right rows
RIGHT JOIN	All right rows + matching left rows
FULL JOIN	All rows from both tables
SELF JOIN	A table joined with itself
CROSS JOIN	Every combination of rows

JOIN → Combine related data from multiple tables.

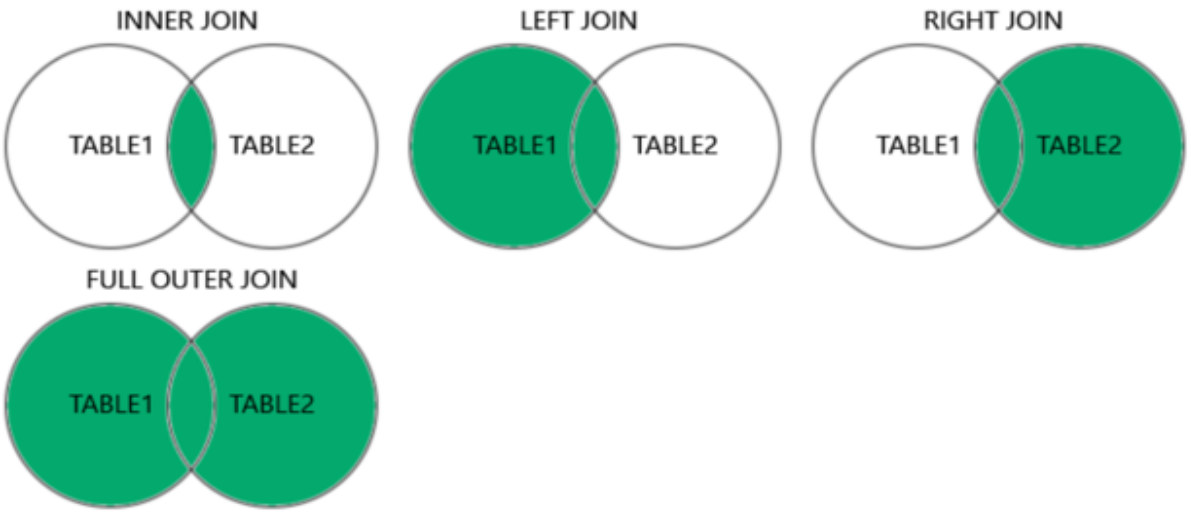


Figure 1: Venn diagram representing how Inner, Left, Right, and Full Outer Joins combine records between tables.

Simple Example

Suppose we have:

Employees

emp_id	emp_name	dept_id
1	Ravi	10
2	Priya	20

Departments

dept_id	dept_name
10	IT
20	HR

The common column is `dept_id`.

```sql
SELECT e.emp_name, d.dept_name
FROM employees e
JOIN departments d
    ON e.dept_id = d.dept_id;
```

Result:

emp_name	dept_name
Ravi	IT
Priya	HR

The `JOIN` connects the tables using a related column.

Quick Understanding

```text
INNER → Matching
LEFT  → Keep Left
RIGHT → Keep Right
FULL  → Keep Both
SELF  → Same Table
CROSS → Every Combination
```

Each join type will be explained in the following topics.

Placement Quick Points

- A `JOIN` combines data from multiple tables.
- Tables are usually joined using related columns.
- `INNER JOIN` returns matching rows.
- `LEFT`, `RIGHT`, and `FULL` joins control which unmatched rows are retained.
- `SELF JOIN` joins a table to itself.
- `CROSS JOIN` creates every possible row combination.

Interview Questions

Q1. What is a JOIN?

> A `JOIN` combines related data from two or more tables.

Q2. Why are joins needed?

> Related data is often stored in separate tables, and joins allow us to retrieve it together.

Q3. What are the common types of joins?

> `text INNER LEFT RIGHT FULL SELF CROSS`

Q4. What column is commonly used to connect tables?

> A related key, such as a primary key and foreign key.

Practice & Hands-On Exercises

Using `employees` and `departments`:

1. Identify the column that connects the two tables.
2. Write a basic `INNER JOIN`:

```
CODE                                                    Copy

SELECT e.emp_id, e.emp_name, d.dept_name
FROM employees e
JOIN departments d
    ON e.dept_id = d.dept_id;
```

1. Explain the difference between `INNER JOIN` and `LEFT JOIN`.
2. List the six common join types.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
TABLE A
   +
TABLE B
   ↓
 JOIN
   ↓
COMBINED DATA
```

Joins allow SQL to combine related data stored in different tables.

End of What are Joins? (Overview)

10.2 INNER JOIN

Level: 10 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`INNER JOIN` returns **only the rows that have a match in both tables**.

For example:

```text
CUSTOMERS
    +
ORDERS
    ↓
INNER JOIN
    ↓
MATCHING RECORDS ONLY
```

> *INNER JOIN → Keep matching rows from both tables.*

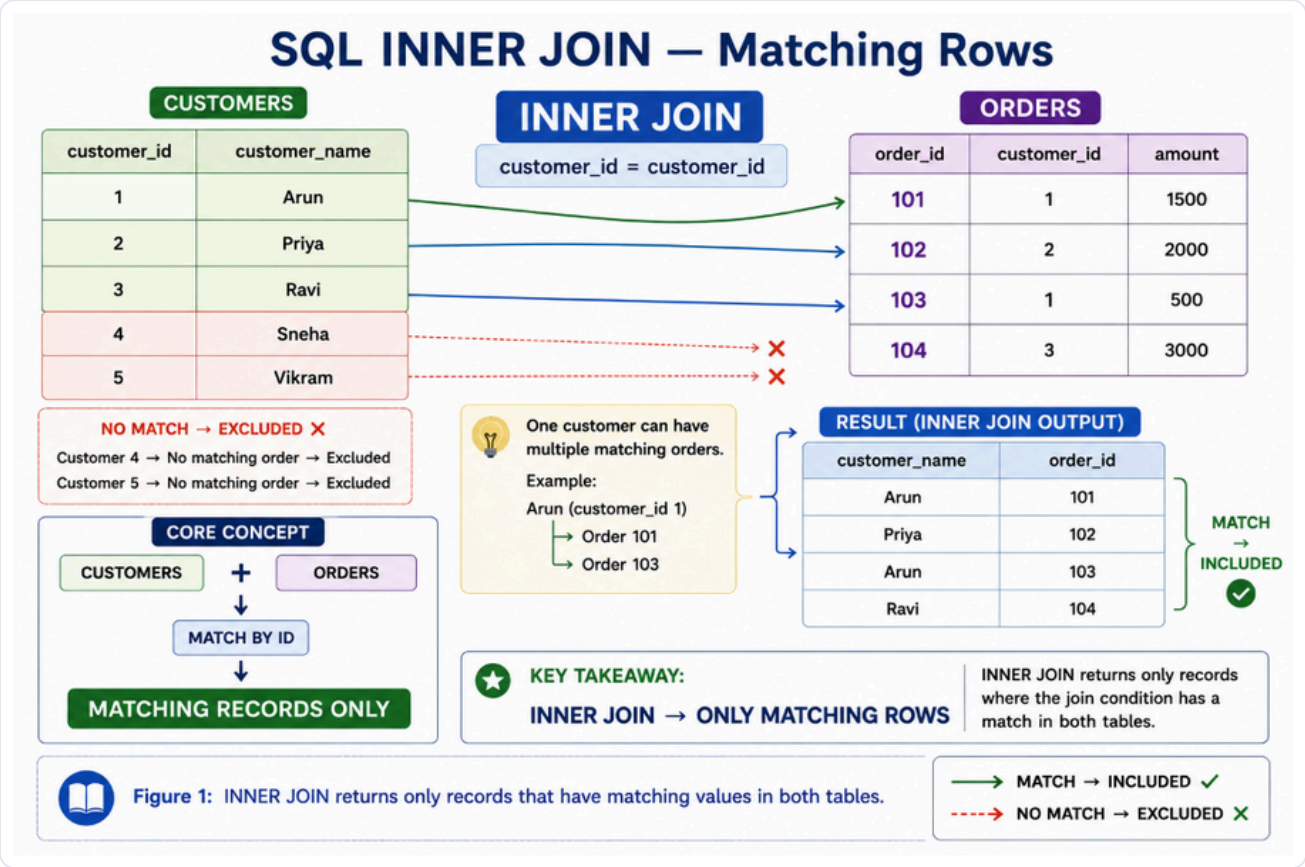


Figure 1: INNER JOIN returns only records that have matching values in both tables.

Practice Tables

This topic uses two tables:

Customers

customer_id	customer_name	city
1	Arun Kumar	Chennai
2	Priya Sharma	Mumbai
3	Ravi Patel	Ahmedabad
4	Sneha Reddy	Hyderabad
5	Vikram Singh	Delhi
6	Meena Iyer	Bangalore
7	Karthik Raj	Chennai
8	Anjali Verma	Pune
9	Rahul Das	Kolkata
10	Divya Nair	Kochi

Orders

order_id	order_date	amount	customer_id
1	2026-01-01	1500	1
2	2026-01-05	2000	2
3	2026-01-07	500	1
4	2026-01-10	3000	3
5	2026-01-12	1200	4
6	2026-01-15	700	5
7	2026-01-18	2200	2
8	2026-01-20	900	6
9	2026-01-22	4000	3
10	2026-01-25	1100	7

Customers `8`, `9`, and `10` have no matching orders.

Basic INNER JOIN

```sql
SELECT customers.customer_name, orders.order_id
FROM customers
INNER JOIN orders
    ON customers.customer_id = orders.customer_id;
```

Result:

customer_name	order_id
Arun Kumar	1
Priya Sharma	2
Arun Kumar	3
Ravi Patel	4
Sneha Reddy	5
Vikram Singh	6
Priya Sharma	7
Meena Iyer	8
Ravi Patel	9
Karthik Raj	10

Only customers with matching orders are returned.

How INNER JOIN Works

The database compares the join columns:

```
TEXT

customers.customer_id
          =
orders.customer_id
```

For example:

```text
Customer ID 1 → Order ID 1
Customer ID 1 → Order ID 3
Customer ID 2 → Order ID 2
```

A customer can therefore appear more than once when that customer has multiple orders.

> *One matching customer can produce multiple result rows.*

Using Table Aliases

Aliases make join queries shorter and easier to read.

```sql
SELECT c.customer_name, o.order_id
FROM customers c
INNER JOIN orders o
    ON c.customer_id = o.customer_id;
```

Here:

```text
c → customers
o → orders
```

INNER JOIN vs No Match

```text
Customers → 8
Orders    → no 8
```

With `INNER JOIN`, customer `8` is not returned.

```text
MATCH → INCLUDED
NO MATCH → EXCLUDED
```

This is the key difference to remember before learning `LEFT JOIN`.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
INNER JOIN returns every row from both tables	It returns only matching rows
One customer always creates one result row	Multiple orders can produce multiple rows
`ON` is optional for every INNER JOIN	A normal join needs a condition unless using a different join form
JOIN combines tables permanently	It creates a result set; it does not merge the physical tables
Customers without orders should appear	INNER JOIN excludes unmatched customers

Placement Quick Points

```text
INNER JOIN
→ MATCHING ROWS ONLY

ON
→ JOIN CONDITION

MATCH
→ INCLUDED

NO MATCH
→ EXCLUDED
```

- `INNER JOIN` combines matching rows from two tables.
- The `ON` clause defines how the tables are matched.
- Unmatched rows are excluded.
- One row from one table can match multiple rows in the other table.
- Aliases make join queries easier to read.

Interview Questions

Q1. What is an INNER JOIN?

An `INNER JOIN` returns rows where the join condition matches in both tables.

Q2. What happens to unmatched rows?

They are excluded from the result.

Q3. Can one customer appear multiple times?

> Yes. If the customer has multiple matching orders, the customer can appear in multiple result rows.

Q4. What is the purpose of ON?

> `ON` specifies the condition used to match rows between the tables.

Q5. Does INNER JOIN modify the original tables?

> No. It produces a query result without changing the source tables.

Q6. What is the difference between JOIN and INNER JOIN?

> `JOIN` is commonly used as shorthand for `INNER JOIN`.

Practice & Hands-On Exercises

Using `customers` and `orders`:

1. Display customer names with their order IDs.
2. Display customer names with order amounts.
3. Find customers who have at least one order.
4. Find customers who placed multiple orders.
5. Explain why customers `8`, `9`, and `10` do not appear.
6. Rewrite the query using table aliases:

```sql
SELECT c.customer_name, o.order_id, o.amount
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id;
```

Copy

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
CUSTOMERS
    +
ORDERS
    ↓
INNER JOIN
    ↓
MATCHING RECORDS
```

> *INNER JOIN returns only the rows that match between the joined tables.*

End of INNER JOIN

10.3 LEFT JOIN

Level: 10 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`LEFT JOIN` returns **all rows from the left table** and the matching rows from the right table.

When no match exists, the right-side columns contain `NULL`.

```text
LEFT TABLE
    +
RIGHT TABLE
    ↓
LEFT JOIN
    ↓
ALL LEFT ROWS
+ MATCHING RIGHT ROWS
```

> *LEFT JOIN → Keep everything from the left table.*

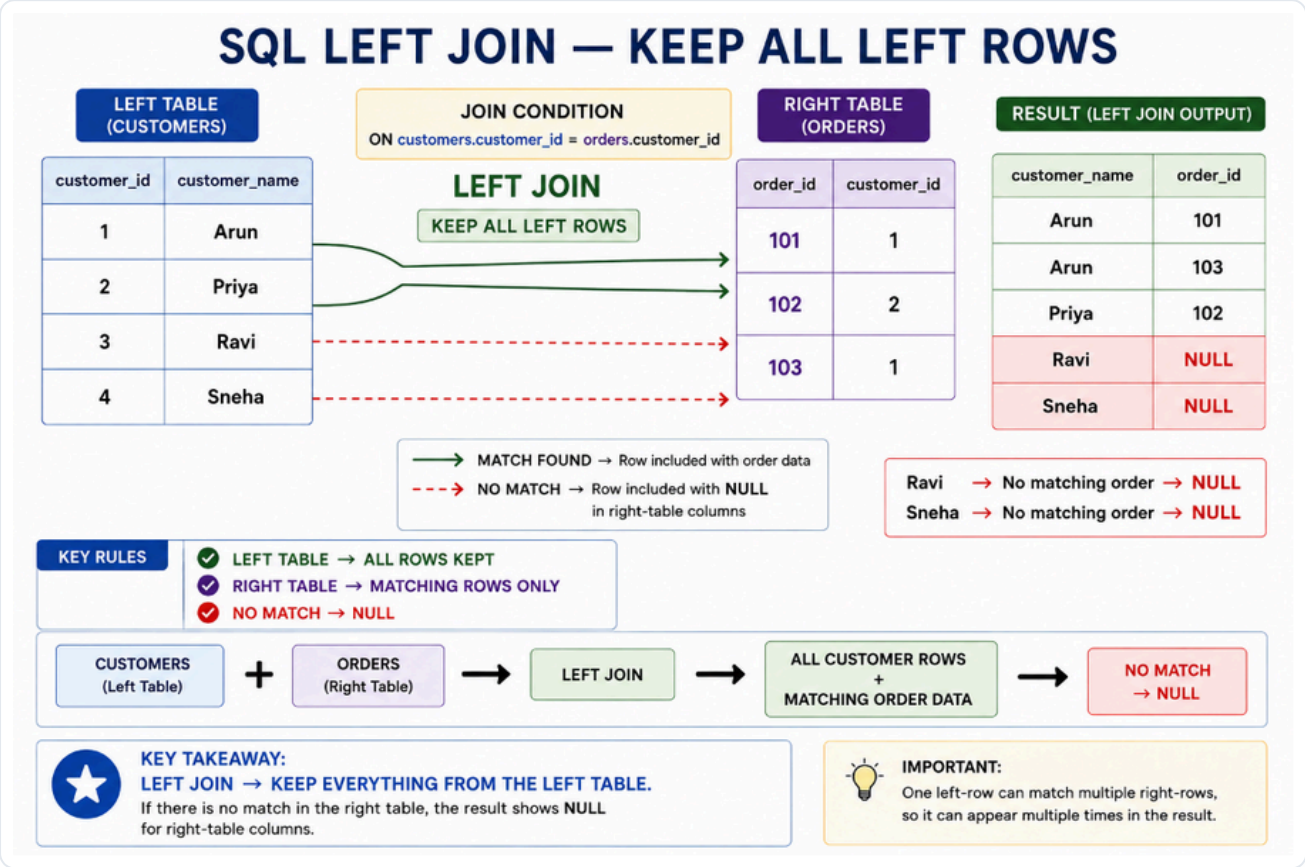


Figure 1: LEFT JOIN keeps every row from the left table and fills right-table columns with NULL when no match exists.

Practice Tables

This topic reuses the `customers` and `orders` tables from **10.2 INNER JOIN**.

Some customers have no orders:

```text
Anjali Verma
Rahul Das
Divya Nair
```

These customers are important for understanding `LEFT JOIN`.

Basic LEFT JOIN

```sql
SELECT customers.customer_name, orders.order_id
FROM customers
LEFT JOIN orders
    ON customers.customer_id = orders.customer_id;
```

Unlike `INNER JOIN`, customers without orders are also returned.

Example:

customer_name	order_id
Arun Kumar	1
Arun Kumar	3
Priya Sharma	2
...	...
Anjali Verma	NULL
Rahul Das	NULL
Divya Nair	NULL

> *No matching order → `NULL` on the right side.*

LEFT JOIN vs INNER JOIN

```text
INNER JOIN
→ MATCHING ROWS ONLY

LEFT JOIN
→ ALL LEFT ROWS
  + MATCHING RIGHT ROWS
```

For the `customers` and `orders` example:

- `INNER JOIN` excludes customers without orders.
- `LEFT JOIN` keeps those customers.

Using Aliases

Aliases make the query easier to read:

```sql
SELECT c.customer_name, o.order_id
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id;
```

Here:

```text
c → customers
o → orders
```

Finding Customers Without Orders

A common use of `LEFT JOIN` is finding rows that have **no match**.

```sql
SELECT c.customer_name
FROM customers c
LEFT JOIN orders o
    ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

This returns customers who have no matching order.

```text
LEFT JOIN
   ↓
KEEP ALL CUSTOMERS
   ↓
WHERE order_id IS NULL
   ↓
CUSTOMERS WITHOUT ORDERS
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
LEFT JOIN keeps all rows from both tables	It keeps all rows from the left table
Unmatched right rows disappear completely	The left row remains and right-side columns become `NULL`
LEFT JOIN and INNER JOIN are the same	INNER JOIN removes unmatched rows
`WHERE o.order_id IS NULL` finds all customers	It finds customers with no matching order in this pattern
RIGHT and LEFT refer to physical table storage	They refer to the position of the tables in the query

Placement Quick Points

```text
LEFT JOIN
→ ALL LEFT ROWS
→ MATCHING RIGHT ROWS

NO MATCH
→ RIGHT SIDE = NULL
```

- `LEFT JOIN` keeps every row from the left table.
- Matching rows from the right table are added.
- If no match exists, right-side columns become `NULL`.

- `LEFT JOIN` is useful for finding records with and without matches.
- `WHERE right_table.key IS NULL` can identify unmatched left rows.

Interview Questions

Q1. What is a LEFT JOIN?

A `LEFT JOIN` returns all rows from the left table and matching rows from the right table.

Q2. What happens when there is no match?

The left row is still returned, and the right-table columns contain `NULL`.

Q3. What is the difference between INNER JOIN and LEFT JOIN?

```
text INNER → Only matching rows LEFT → All left rows + matching right rows
```

Q4. How do you find customers without orders?

Use a `LEFT JOIN` and check the right-side key for `NULL`:

```
Sql WHERE o.order_id IS NULL;
```

Q5. Which table is the left table?

The table written immediately after `FROM`.

Practice & Hands-On Exercises

Using `customers` and `orders`:

1. Display all customers with their order IDs:

```
CODE
Copy

SELECT c.customer_name, o.order_id
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

1. Identify customers who have no orders:

```
CODE
Copy

SELECT c.customer_name
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```

1. Display customer names and order amounts.
2. Rewrite the query using aliases.
3. Explain the difference between `INNER JOIN` and `LEFT JOIN`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE
Copy

LEFT JOIN
    ↓
ALL LEFT ROWS
    +
MATCHING RIGHT ROWS
    ↓
NO MATCH → NULL
```

> ***LEFT JOIN keeps every row from the left table and adds matching data from the right table; unmatched right-side values become NULL.***

End of LEFT JOIN

10.4 RIGHT JOIN

Level: 10 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`RIGHT JOIN` returns **all rows from the right table** and matching rows from the left table.

When no match exists, the left-table columns contain `NULL`.

```text
LEFT TABLE
     +
RIGHT TABLE
     ↓
RIGHT JOIN
     ↓
ALL RIGHT ROWS
+ MATCHING LEFT ROWS
```

> *RIGHT JOIN → Keep everything from the right table.*

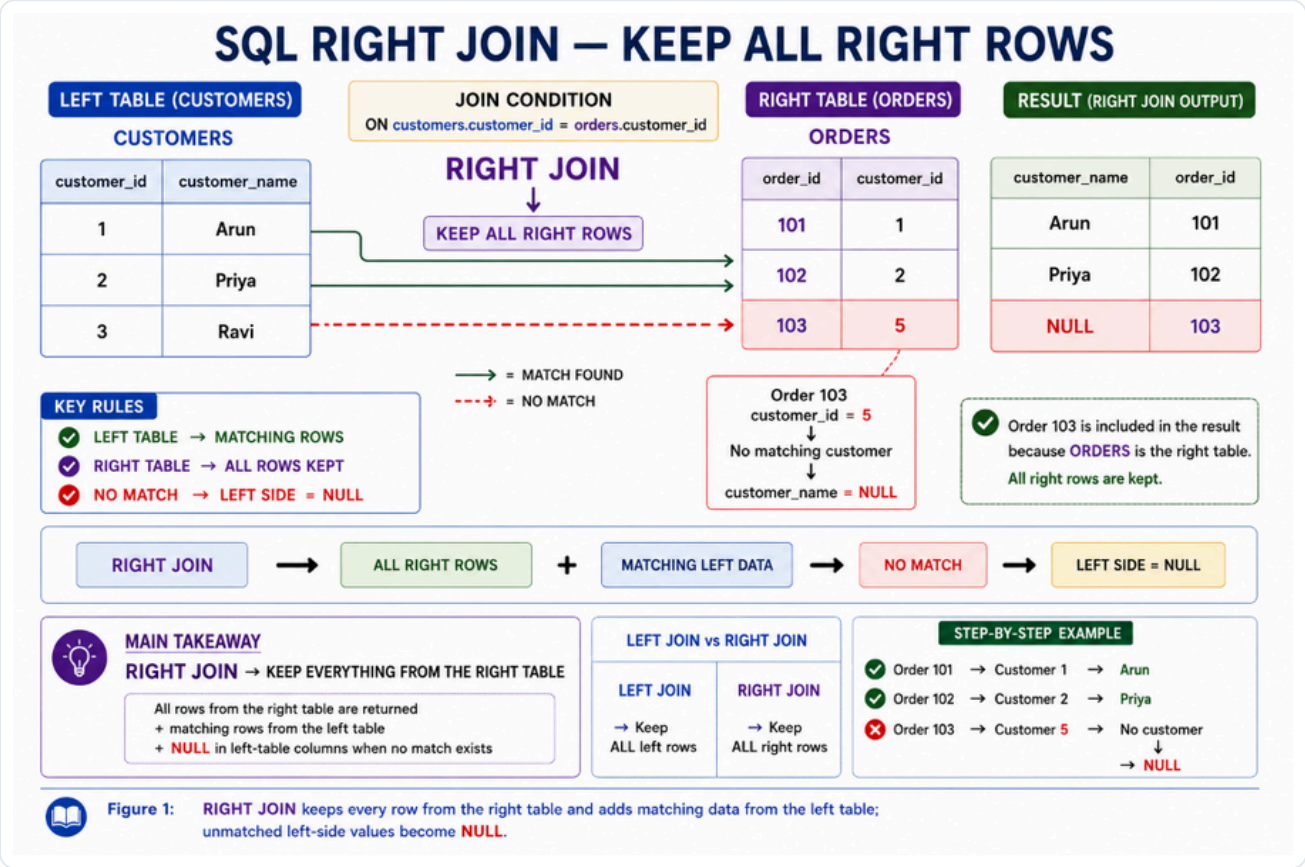


Figure 1: RIGHT JOIN keeps every row from the right table and adds matching data from the left table; unmatched left-side values become NULL.

Practice Tables

This topic reuses the `customers` and `orders` tables from **10.2 INNER JOIN**.

For the original practice data, every order has a matching customer. Therefore, the basic `RIGHT JOIN` produces the same rows as the `INNER JOIN`.

To clearly demonstrate the difference, consider an order whose customer does not exist:

```text
order_id = 11
customer_id = 99
```

Basic RIGHT JOIN

```sql
SELECT c.customer_name, o.order_id
FROM customers c
RIGHT JOIN orders o
    ON c.customer_id = o.customer_id;
```

The important rule is:

```text
RIGHT TABLE → ALL ROWS KEPT
LEFT TABLE  → MATCHING ROWS ONLY
```

If an order has no matching customer, `customer_name` becomes `NULL`.

Example with No Match

Suppose the `orders` table contains:

order_id	customer_id
10	7
11	99

Customer `99` does not exist.

The result is:

customer_name	order_id
Karthik Raj	10
NULL	11

The order is still returned because `orders` is the **right table**.

> *No matching customer → left-side values become `NULL`.*

RIGHT JOIN vs LEFT JOIN

These two joins are mirror images.

```text
LEFT JOIN
→ Keep ALL LEFT rows

RIGHT JOIN
→ Keep ALL RIGHT rows
```

For example:

```sql
FROM customers c
LEFT JOIN orders o ...
```

keeps all customers.

While:

```
SQL
Copy

FROM customers c
RIGHT JOIN orders o ...
```

keeps all orders.

In practice, many developers prefer `LEFT JOIN` because reversing the table order can often make the query easier to read.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
RIGHT JOIN keeps all left rows	It keeps all right rows
Unmatched right rows disappear	They remain, with `NULL` on the left
RIGHT JOIN is different from LEFT JOIN in logic	They are mirror-image concepts
The right table is always the physically stored/rightmost database object	"Right" means the table written after `RIGHT JOIN`
RIGHT JOIN always produces more rows than INNER JOIN	Only when unmatched right-side rows exist

Placement Quick Points

```text
RIGHT JOIN
→ ALL RIGHT ROWS

MATCH
→ ADD LEFT DATA

NO MATCH
→ LEFT SIDE = NULL
```

Copy

- `RIGHT JOIN` keeps every row from the right table.
- Matching rows from the left table are added.
- If no match exists, left-side columns contain `NULL`.
- It is the mirror image of `LEFT JOIN`.
- The difference becomes visible when the right table contains unmatched rows.

Interview Questions

Q1. What is a RIGHT JOIN?

A `RIGHT JOIN` returns all rows from the right table and matching rows from the left table.

Q2. What happens when there is no match?

The right-side row remains, and the left-table columns contain `NULL`.

Q3. What is the difference between LEFT JOIN and RIGHT JOIN?

```text LEFT JOIN → Keep all left rows RIGHT JOIN → Keep all right rows```
```

Q4. When does RIGHT JOIN differ from INNER JOIN?

> When the right table contains rows that have no matching row in the left table.

Q5. Which table is the right table?

> The table written immediately after `RIGHT JOIN`.

Practice & Hands-On Exercises

Using `customers` and `orders`:

1. Write a basic `RIGHT JOIN`:

```
CODE
Copy

SELECT c.customer_name, o.order_id
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;
```

1. Display customer names with order IDs.
2. Explain which table's rows are always preserved.
3. Add an order with a non-existing `customer_id` and observe the result.
4. Compare the result with an `INNER JOIN`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE

Copy

```
RIGHT JOIN
     ↓
ALL RIGHT ROWS
     +
MATCHING LEFT ROWS
     ↓
NO MATCH → NULL
```

> ***RIGHT JOIN keeps all rows from the right table and fills the left-side columns with NULL when no match exists.***

End of RIGHT JOIN

10.5 FULL JOIN

Level: 10 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`FULL JOIN` returns **all rows from both tables**.

- Matching rows are combined.
- Unmatched left rows remain with `NULL` on the right.
- Unmatched right rows remain with `NULL` on the left.

```text
LEFT TABLE
     +
RIGHT TABLE
     ↓
FULL JOIN
     ↓
ALL ROWS FROM BOTH
```

> *FULL JOIN → Keep everything from both tables.*

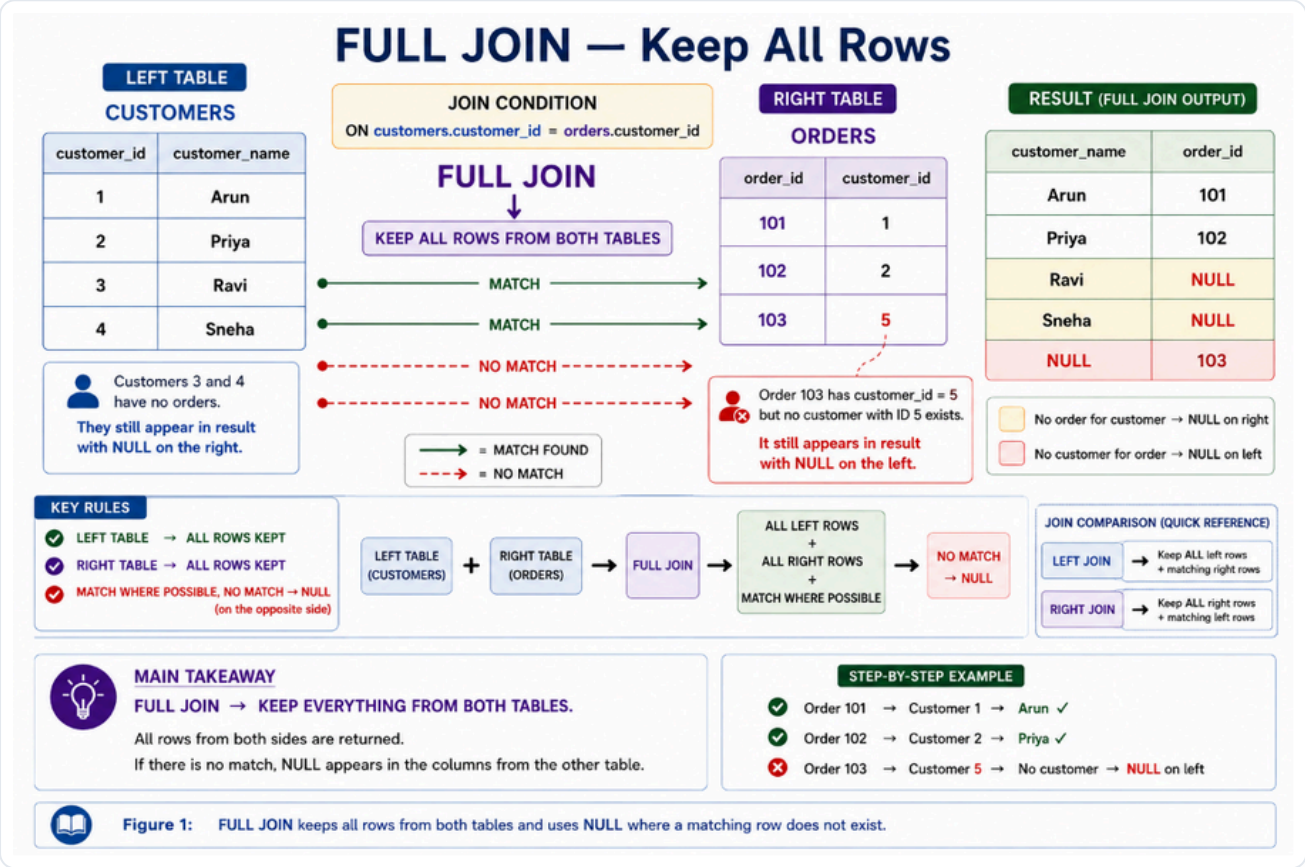


Figure 1: FULL JOIN keeps all rows from both tables and uses NULL where a matching row does not exist.

Practice Tables

This topic reuses the `customers` and `orders` tables from **10.2**.

In the original data:

- Some customers have no orders.
- Every existing order has a matching customer.

For example:

```
TEXT

Anjali Verma → no order
Rahul Das    → no order
Divya Nair   → no order
```

Basic FULL JOIN

```sql
SQL

SELECT c.customer_name, o.order_id
FROM customers c
FULL JOIN orders o
    ON c.customer_id = o.customer_id;
```

A `FULL JOIN` keeps:

```
TEXT

Matching customers + orders
     +
Customers without orders
     +
Orders without customers
```

Understanding the Result

Suppose:

```
TEXT

Customer 1 → Order 101
Customer 2 → Order 102
Customer 3 → No order
Order 103  → No customer
```

The result is:

customer_name	order_id
Customer 1	101
Customer 2	102
Customer 3	NULL
NULL	103

The unmatched row gets `NULL` on the side where no match exists.

> ***FULL JOIN preserves unmatched rows from both sides.***

FULL JOIN vs Other Joins

```text
INNER JOIN → MATCHING ONLY

LEFT JOIN
→ ALL LEFT + MATCHING RIGHT

RIGHT JOIN
→ ALL RIGHT + MATCHING LEFT

FULL JOIN
→ ALL LEFT + ALL RIGHT
```

Copy

This is the easiest way to remember the difference.

Important Note About the Extra Order

The original `orders` table has a foreign key:

```sql
customer_id INT REFERENCES customers(customer_id)
```

So this would normally fail:

```sql
INSERT INTO orders
VALUES (11, '2026-01-30', 9999, 999);
```

because customer `999` does not exist.

You do **not** need to break the foreign-key constraint just to understand `FULL JOIN`. The unmatched customers already demonstrate the part where the left side has no match.

To demonstrate an **unmatched right row**, you would need a dataset where the relationship is not enforced, such as a separate practice table without that foreign key.

Finding Unmatched Rows

A `FULL JOIN` can help identify records that exist on only one side.

Conceptually:

```text
LEFT ONLY
   +
MATCHED
   +
RIGHT ONLY
```

Copy

This makes `FULL JOIN` useful for comparing two datasets and finding missing records.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
FULL JOIN returns matching rows only	It returns all rows from both tables
NULL means the row was removed	The row remains; NULL indicates no match on the other side
FULL JOIN is the same as INNER JOIN	INNER JOIN excludes unmatched rows
FULL JOIN always needs unmatched rows to work	It still returns all rows even when everything matches
RIGHT and LEFT tables mean physical database positions	They refer to the tables' positions in the JOIN statement

Placement Quick Points

```text
FULL JOIN
→ ALL LEFT ROWS
→ ALL RIGHT ROWS
→ MATCH WHERE POSSIBLE
→ NO MATCH = NULL
```

- `FULL JOIN` keeps every row from both tables.
- Matching rows are combined.
- Unmatched rows remain.
- Missing values appear as `NULL` on the opposite side.
- It is useful for comparing two datasets and finding unmatched records.

Interview Questions

Q1. What is a FULL JOIN?

A `FULL JOIN` returns all rows from both tables, matching rows where possible.

Q2. What happens when there is no match?

The row is still returned, and the columns from the other table contain `NULL`.

Q3. What is the difference between INNER JOIN and FULL JOIN?

`text INNER → Matching rows only FULL → All rows from both tables`

Q4. When is FULL JOIN useful?

It is useful when you need to see **all records from both sides**, including unmatched records.

Q5. Can FULL JOIN show NULL values?

> Yes. Unmatched rows contain `NULL` for columns from the other table.

Practice & Hands-On Exercises

Using `customers` and `orders`:

1. Write a `FULL JOIN`:

```
CODE
Copy

SELECT c.customer_name, o.order_id
FROM customers c
FULL JOIN orders o
    ON c.customer_id = o.customer_id;
```

1. Display customer names and order IDs.
2. Identify customers without orders.
3. Explain where `NULL` appears in the result.
4. Compare `FULL JOIN` with `INNER JOIN` and `LEFT JOIN`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE

Copy

```
FULL JOIN
    ↓
ALL LEFT ROWS
+
ALL RIGHT ROWS
    ↓
MATCH WHEN POSSIBLE
    ↓
NO MATCH → NULL
```

FULL JOIN keeps all rows from both tables, combining matches and preserving unmatched rows with NULL values.

End of FULL JOIN

10.6 CROSS JOIN

Level: 10 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`CROSS JOIN` returns **every possible combination of rows** from two tables.

It does not use an `ON` condition.

```text
Table A × Table B
       ↓
Every possible combination
```

If Table A has **2 rows** and Table B has **3 rows**:

```text
2 × 3 = 6 rows
```

> *CROSS JOIN → Every row paired with every row.*

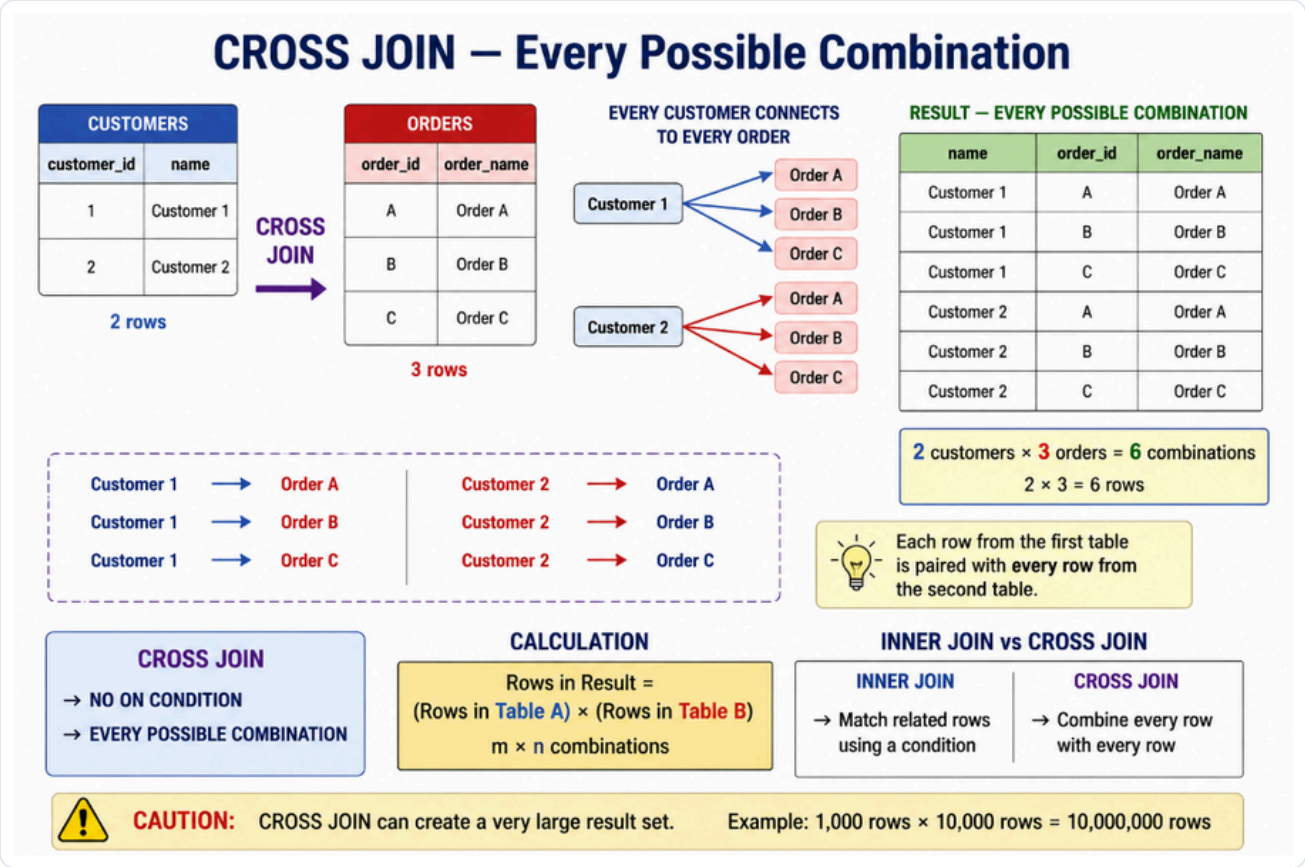


Figure 1: CROSS JOIN produces a Cartesian product by pairing every row from one table with every row from the other table.

Practice Tables

Customers

id	name
1	Michael
2	Sophia

Orders

order_id	amount
101	999
102	4999

Basic CROSS JOIN

```sql
SELECT c.name, o.order_id, o.amount
FROM customers_test c
CROSS JOIN orders_test o;
```

With 2 customers and 2 orders:

```text
2 × 2 = 4 rows
```

Result:

name	order_id	amount
Michael	101	999
Michael	102	4999
Sophia	101	999
Sophia	102	4999

Notice that the query **does not check whether the customer actually placed the order**.

CROSS JOIN vs INNER JOIN

```text
INNER JOIN
→ MATCH RELATED ROWS

CROSS JOIN
→ COMBINE EVERY ROW
```

`CROSS JOIN` is useful when you genuinely need all combinations, such as generating possible product/size combinations or test scenarios.

Common Beginner Mistake

A `CROSS JOIN` can produce a very large result.

For example:

```text
1,000 rows × 10,000 rows
= 10,000,000 rows
```

So it should be used carefully.

> *CROSS JOIN can cause a result-set explosion.*

Placement Quick Points

- `CROSS JOIN` produces every possible row combination.
- It does not require an `ON` condition.
- Result size = **rows in Table A × rows in Table B**.
- It is also called a **Cartesian product**.
- Large tables can produce extremely large results.

Interview Questions

Q1. What is a CROSS JOIN?

A `CROSS JOIN` returns every possible combination of rows from two tables.

Q2. Does CROSS JOIN use ON?

No.

Q3. How many rows are produced?

If Table A has `m` rows and Table B has `n` rows:

`text m × n`

Q4. What is a Cartesian product?

It is the result containing every possible combination of rows from two sets/tables.

Q5. What is a common risk of CROSS JOIN?

It can produce a very large result set when both tables contain many rows.

Practice & Hands-On Exercises

1. Write a `CROSS JOIN` between `customers_test` and `orders_test`:

```text
SELECT c.name, o.order_id, o.amount
FROM customers_test c
CROSS JOIN orders_test o;
```

Copy

1. Count the number of rows produced.
2. Explain why 2 × 2 produces 4 rows.
3. Explain the difference between `CROSS JOIN` and `INNER JOIN`.
4. What would happen with 100 customers and 1,000 products?

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```sql
CROSS JOIN
    ↓
EVERY ROW × EVERY ROW
    ↓
m × n COMBINATIONS
```

Copy

> ***CROSS JOIN creates a Cartesian product by pairing every row from the first table with every row from the second table.***

End of CROSS JOIN

10.7 SELF JOIN

Level: 10 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **SELF JOIN** joins a table with itself.

It is useful when rows in the same table are related to each other, such as an **employee and their manager**.

```text
EMPLOYEES
   ↓
SELF JOIN
   ↓
EMPLOYEE ⇔ MANAGER
```

> *SELF JOIN → Join a table to itself.*

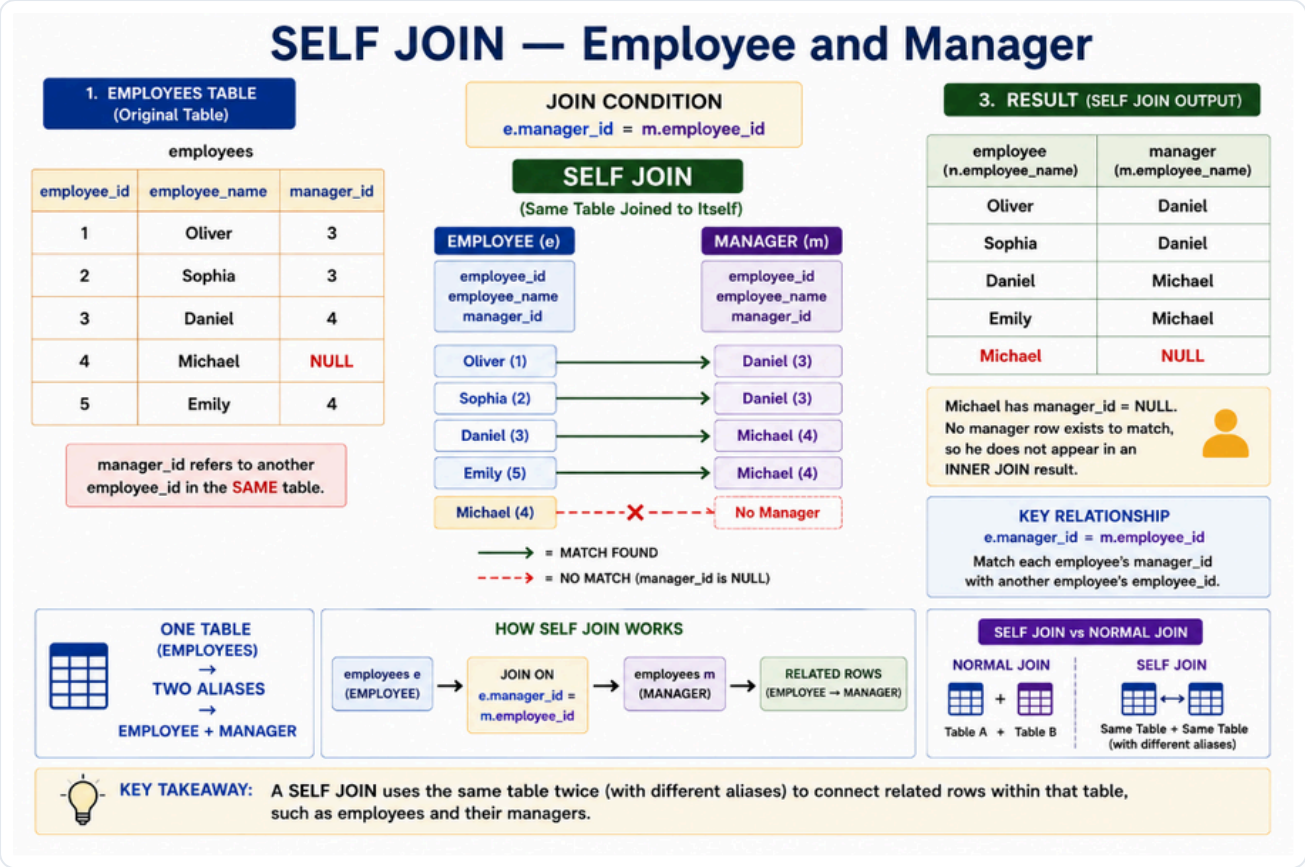


Figure 1: A SELF JOIN uses two aliases of the same table to connect related rows, such as employees and their managers.

Practice Table

```sql
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(50),
    manager_id INT REFERENCES employees(employee_id)
);

INSERT INTO employees
(employee_id, employee_name, manager_id)
VALUES
(1, 'Oliver', 3),
(2, 'Sophia', 3),
(3, 'Daniel', 4),
(4, 'Michael', NULL),
(5, 'Emily', 4);
```

Here, `manager_id` refers to another `employee_id` in the **same table**.

Example: Employee and Manager

```sql
SELECT
    e.employee_name AS employee,
    m.employee_name AS manager
FROM employees e
JOIN employees m
    ON e.manager_id = m.employee_id;
```

Result:

employee	manager
Oliver	Daniel
Sophia	Daniel
Daniel	Michael
Emily	Michael

Here:

```
TEXT

e → employee
m → manager
```

Both aliases refer to the same `employees` table.

How SELF JOIN Works

The relationship is:

```
TEXT

employees e
     |
     | e.manager_id
     ↓
employees m
     |
     | m.employee_id
```

The join condition:

```
SQL

e.manager_id = m.employee_id
```

matches each employee's manager ID with another employee's ID.

Michael has `manager_id = NULL`, so he has no manager to match.

SELF JOIN vs Normal JOIN

```text
Normal JOIN
→ Table A + Table B

SELF JOIN
→ Same Table + Same Table
```

The table is not duplicated physically. **Aliases allow us to treat it as two logical instances in the query.**

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
SELF JOIN requires two physical tables	It uses the same table twice in the query
`e` and `m` are separate tables	They are aliases of the same table
SELF JOIN always returns every row	Unmatched rows can be excluded with an inner join
`manager_id` must always have a value	It can be `NULL` for employees without a manager
SELF JOIN creates duplicate table data	It only combines rows in the query result

Placement Quick Points

```text
SELF JOIN
→ SAME TABLE TWICE

e
→ EMPLOYEE

m
→ MANAGER

e.manager_id = m.employee_id
→ MATCH RELATED ROWS
```

- A `SELF JOIN` joins a table with itself.

- Table aliases distinguish the two logical roles.
- It is useful for hierarchical data such as employee-manager relationships.
- The same table does not need to be physically duplicated.
- `NULL` manager IDs represent employees without a matching manager in this example.

Interview Questions

Q1. What is a SELF JOIN?

> A SELF JOIN joins a table with itself to compare or connect rows within the same table.

Q2. Why are aliases needed?

> Aliases allow the same table to be referenced as two different logical roles.

Q3. Give a real-world example of SELF JOIN.

> An employee-manager hierarchy is a common example.

Q4. Does SELF JOIN require two physical tables?

> No. It uses the same table twice in the query.

Q5. Why doesn't Michael appear in the INNER JOIN result?

> His `manager_id` is `NULL`, so there is no manager row to match.

Practice & Hands-On Exercises

Using the `employees` table:

1. Display each employee with their manager:

```sql
SELECT e.employee_name AS employee, m.employee_name AS manager
FROM employees e
JOIN employees m ON e.manager_id = m.employee_id;
```

1. Find employees who have a manager.
2. Explain why Michael is not returned by the `INNER JOIN`.
3. Identify the two aliases used in the query.
4. Explain the join condition `e.manager_id = m.employee_id`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
SAME TABLE
   ↓
TWO ALIASES
   ↓
EMPLOYEE ↔ MANAGER
```

> ***A SELF JOIN uses the same table with different aliases to connect related rows within that table.***

End of SELF JOIN

11.1 GROUP BY

Level: 11 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`GROUP BY` is used to **group rows with the same value** and create summary results.

It is commonly used with aggregate functions such as `COUNT()`, `SUM()`, and `AVG()`.

```text
ROWS
 ↓
GROUP BY
 ↓
GROUPS
 ↓
SUMMARY
```

> *GROUP BY → Group similar rows and calculate a summary for each group.*

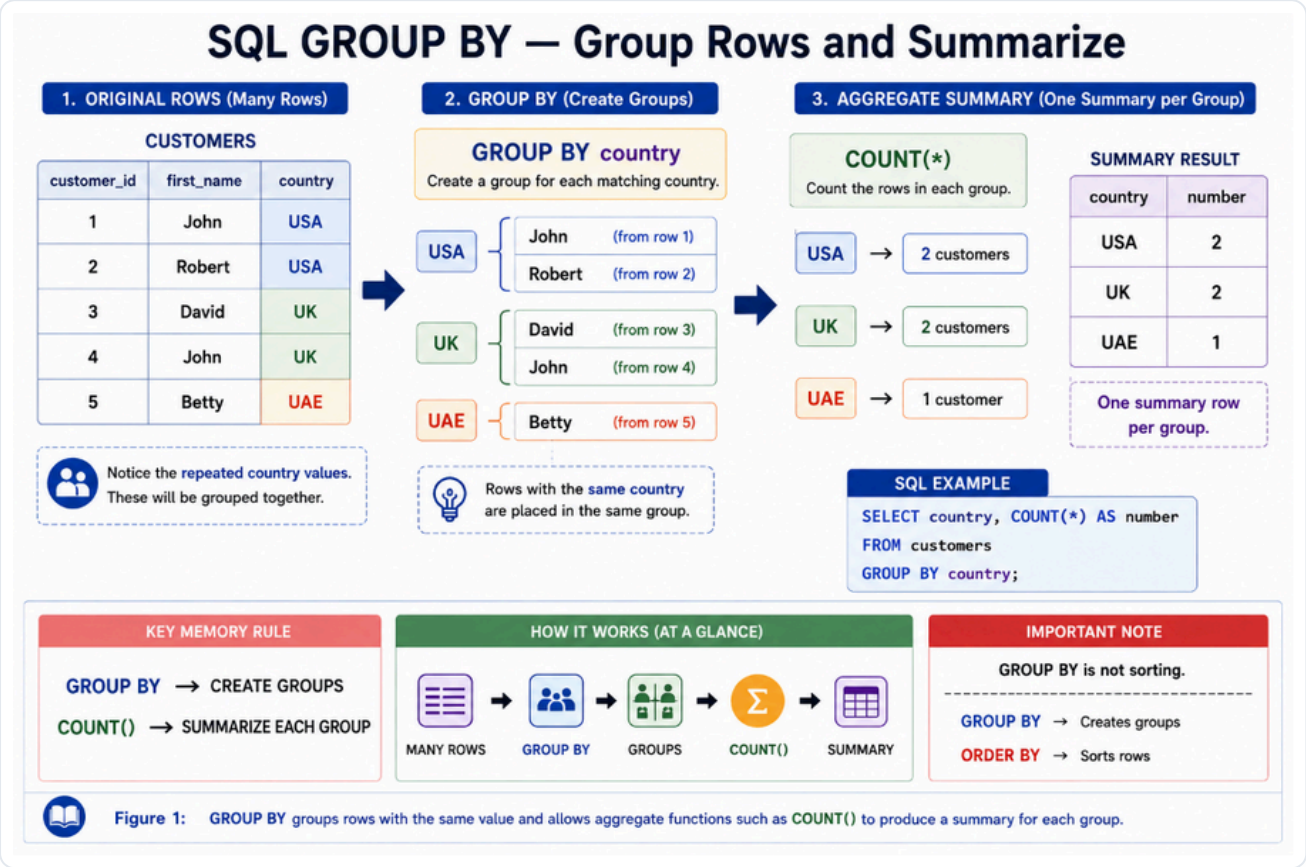


Figure 1: GROUP BY groups rows with the same value and allows aggregate functions such as COUNT() to produce a summary for each group.

Practice Table

This topic uses the `customers` table:

customer_id	first_name	country
1	John	USA
2	Robert	USA
3	David	UK
4	John	UK
5	Betty	UAE

Basic GROUP BY

Find the number of customers in each country:

```sql
SELECT country, COUNT(*) AS number
FROM customers
GROUP BY country;
```

Result:

country	number
UAE	1
UK	2
USA	2

Instead of returning every customer row, SQL creates **one summary row for each country**.

GROUP BY with Other Aggregates

`GROUP BY` can be combined with aggregate functions.

For example:

```sql
SELECT country, AVG(age) AS average_age
FROM customers
GROUP BY country;
```

This calculates the average age separately for each country.

```text
GROUP BY country
        ↓
One group per country
        ↓
AVG(age) for each group
```

How GROUP BY Works

Suppose the countries are:

```text
USA
USA
UK
UK
UAE
```

`GROUP BY country` creates:

```
TEXT

USA → 2 rows
UK  → 2 rows
UAE → 1 row
```

Then an aggregate function can summarize each group.

> *GROUP BY does not simply sort rows. It creates groups based on matching values.*

GROUP BY vs SELECT

A common beginner confusion is:

```
TEXT

SELECT country
→ Show country

GROUP BY country
→ Create one group for each country
```

For example:

```sql
SQL

SELECT country, COUNT(*)
FROM customers
GROUP BY country;
```

`COUNT(*)` is calculated separately for every country group.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`GROUP BY` sorts the result	`ORDER BY` sorts; `GROUP BY` creates groups
`GROUP BY` returns every original row	It usually produces one result row per group when used with aggregates
`GROUP BY` can be used only with `COUNT()`	It can be used with `SUM()`, `AVG()`, `MIN()`, `MAX()`, and others
Each group contains only one row	A group can contain many rows
`GROUP BY country` counts automatically	An aggregate such as `COUNT()` is needed to calculate a count

Placement Quick Points

```text
GROUP BY
→ CREATE GROUPS

COUNT()
→ COUNT EACH GROUP

SUM()
→ TOTAL EACH GROUP

AVG()
→ AVERAGE EACH GROUP
```

- `GROUP BY` groups rows with the same value.

- It is commonly used with aggregate functions.
- Each group can produce a summary result.
- `COUNT()`, `SUM()`, `AVG()`, `MIN()`, and `MAX()` can be used with groups.
- `GROUP BY` is different from `ORDER BY`.

Interview Questions

Q1. What is GROUP BY?

`GROUP BY` groups rows with the same values so that aggregate calculations can be performed for each group.

Q2. Why is GROUP BY commonly used with aggregate functions?

It allows calculations such as count, total, or average to be performed separately for each group.

Q3. What is the difference between GROUP BY and ORDER BY?

```
text GROUP BY → Group rows ORDER BY → Sort rows
```

Q4. Can GROUP BY be used with COUNT()?

Yes.

```
Sql SELECT country, COUNT(*) FROM customers GROUP BY country;
```

Q5. Can GROUP BY be used with AVG()?

Yes.

```
Sql SELECT country, AVG(age) FROM customers GROUP BY country;
```

Practice & Hands-On Exercises

Using the `customers` table:

1. Count customers in each country:

```
CODE
Copy
SELECT country, COUNT(*) FROM customers GROUP BY country;
```

1. Find the average age for each country:

```
CODE
Copy
SELECT country, AVG(age) FROM customers GROUP BY country;
```

1. Find the minimum age in each country:

```
CODE
Copy
SELECT country, MIN(age) FROM customers GROUP BY country;
```

1. Find the maximum age in each country:

```
CODE
Copy
SELECT country, MAX(age) FROM customers GROUP BY country;
```

1. Explain the difference between `GROUP BY` and `ORDER BY`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE

Copy

```
ROWS
 ↓
GROUP BY
 ↓
GROUPS
 ↓
AGGREGATE
 ↓
SUMMARY
```

GROUP BY creates groups of similar rows so aggregate functions can calculate a summary for each group.

End of GROUP BY

11.2 HAVING (vs WHERE)

Level: 11 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`HAVING` is used to **filter groups after** `GROUP BY`.

The key difference is:

```text
WHERE
→ FILTER ROWS

GROUP BY
→ CREATE GROUPS

HAVING
→ FILTER GROUPS
```

> *WHERE filters individual rows. HAVING filters grouped results.*

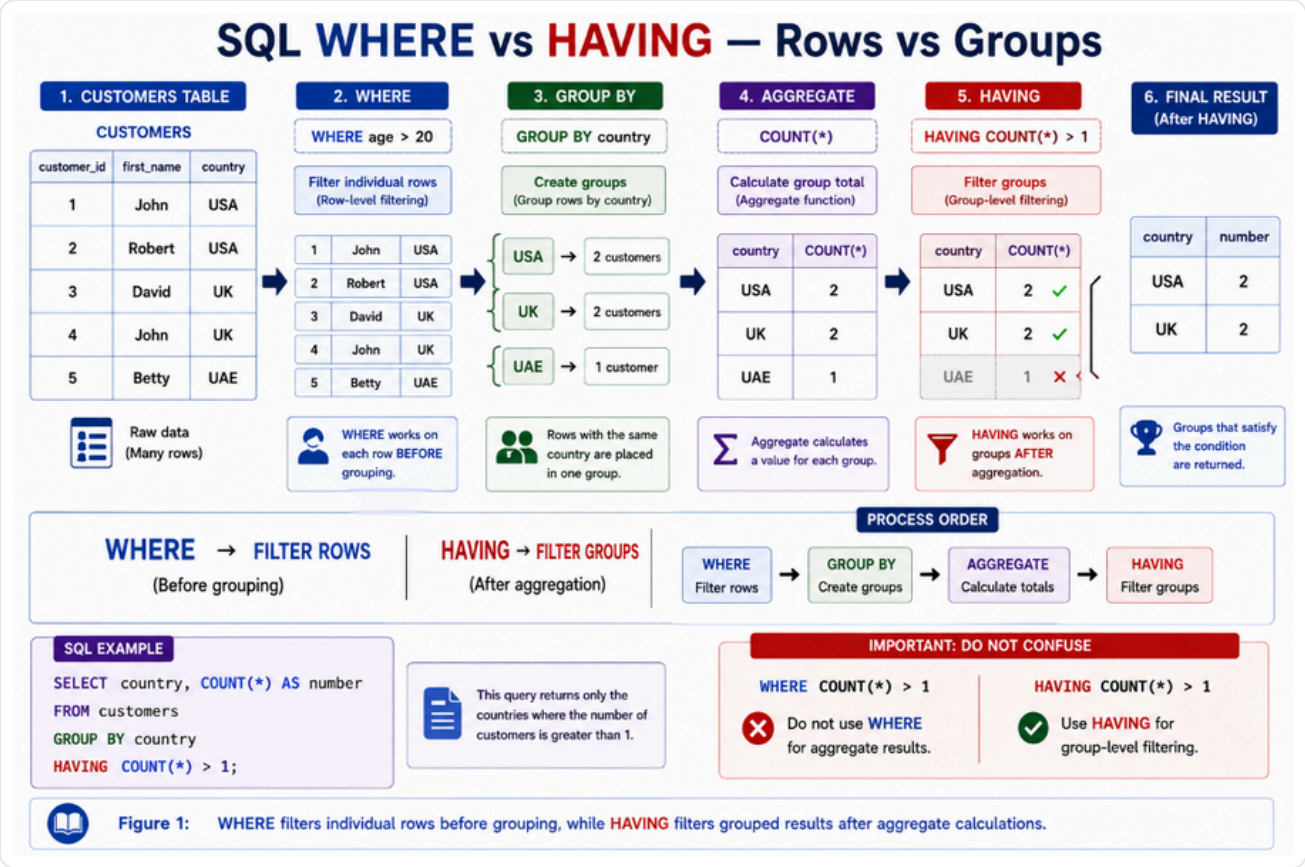


Figure 1: WHERE filters individual rows before grouping, while HAVING filters groups after aggregate calculations.

Basic HAVING

Using the `customers` table:

```sql
SELECT country, COUNT(*) AS number
FROM customers
GROUP BY country
HAVING COUNT(*) > 1;
```

Result:

country	number
USA	2
UK	2

`UAE` is removed because its count is only `1`.

WHERE vs HAVING

```text
WHERE
→ Filter individual rows
→ Used before GROUP BY

HAVING
→ Filter groups
→ Used after GROUP BY
```

For example:

```sql
SELECT country, COUNT(*)
FROM customers
WHERE age > 20
GROUP BY country
HAVING COUNT(*) > 1;
```

Here:

- `WHERE age > 20` filters rows first.
- `GROUP BY country` creates country groups.
- `HAVING COUNT(*) > 1` filters those groups.

Important Point

`WHERE` cannot directly filter an aggregate result such as:

```sql
WHERE COUNT(*) > 1
```

Use:

```sql
HAVING COUNT(*) > 1
```

> *WHERE → Row-level condition HAVING → Group-level condition*

Simple Flow

```text
TABLE
 ↓
WHERE
 ↓
GROUP BY
 ↓
AGGREGATE
 ↓
HAVING
 ↓
RESULT
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`WHERE` and `HAVING` do the same thing	WHERE filters rows; HAVING filters groups
HAVING is used before GROUP BY	HAVING filters grouped results
`WHERE COUNT(*) > 1` is valid	Use `HAVING COUNT(*) > 1`
HAVING can only be used with `COUNT()`	It can filter results using aggregate expressions such as `SUM()`, `AVG()`, `MIN()`, and `MAX()`
HAVING filters the original table permanently	It only filters the query result

Placement Quick Points

```text
WHERE
→ FILTER ROWS

GROUP BY
→ CREATE GROUPS

HAVING
→ FILTER GROUPS
```

- `WHERE` filters individual rows.
- `GROUP BY` creates groups.

- Aggregate functions calculate values for each group.
- `HAVING` filters the grouped results.
- `WHERE` cannot directly filter an aggregate calculation such as `COUNT(*)`.

Interview Questions

Q1. What is HAVING?

`HAVING` filters groups created by `GROUP BY`.

Q2. What is the difference between WHERE and HAVING?

```
text WHERE → Filter rows HAVING → Filter groups
```

Q3. Why can't WHERE be used with COUNT()?

Because `WHERE` operates on individual rows before grouping, while `COUNT()` is calculated for groups. `HAVING` is used to filter the aggregate result.

Q4. Can HAVING be used without GROUP BY?

Some database systems allow it in specific cases, but for beginner SQL, understand `HAVING` primarily as a group-filtering clause used with aggregation.

Q5. Can HAVING use SUM() or AVG()?

Yes.

```
Sql HAVING SUM(salary) > 100000
```

Practice & Hands-On Exercises

Using `customers`:

1. Find countries having more than one customer:

```
CODE
Copy

SELECT country, COUNT(*) AS number
FROM customers
GROUP BY country
HAVING COUNT(*) > 1;
```

1. Find countries having more than one customer after filtering `age > 20`:

```
CODE
Copy

SELECT country, COUNT(*)
FROM customers
WHERE age > 20
GROUP BY country
HAVING COUNT(*) > 1;
```

1. Explain the difference between `WHERE` and `HAVING`.
2. Rewrite a query using `HAVING COUNT(*) > 1`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE

Copy

```
WHERE
→ FILTER ROWS

GROUP BY
→ GROUP ROWS

HAVING
→ FILTER GROUPS
```

Use WHERE to filter rows before grouping and HAVING to filter groups after aggregation.

End of HAVING (vs WHERE)

11.3 UNION / UNION ALL

Level: 11 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`UNION` and `UNION ALL` combine the results of two or more `SELECT` queries into one result.

Operator	Purpose
`UNION`	Combines results and removes duplicates
`UNION ALL`	Combines results and keeps duplicates

> ***UNION → Combine + Remove duplicates UNION ALL → Combine + Keep duplicates***

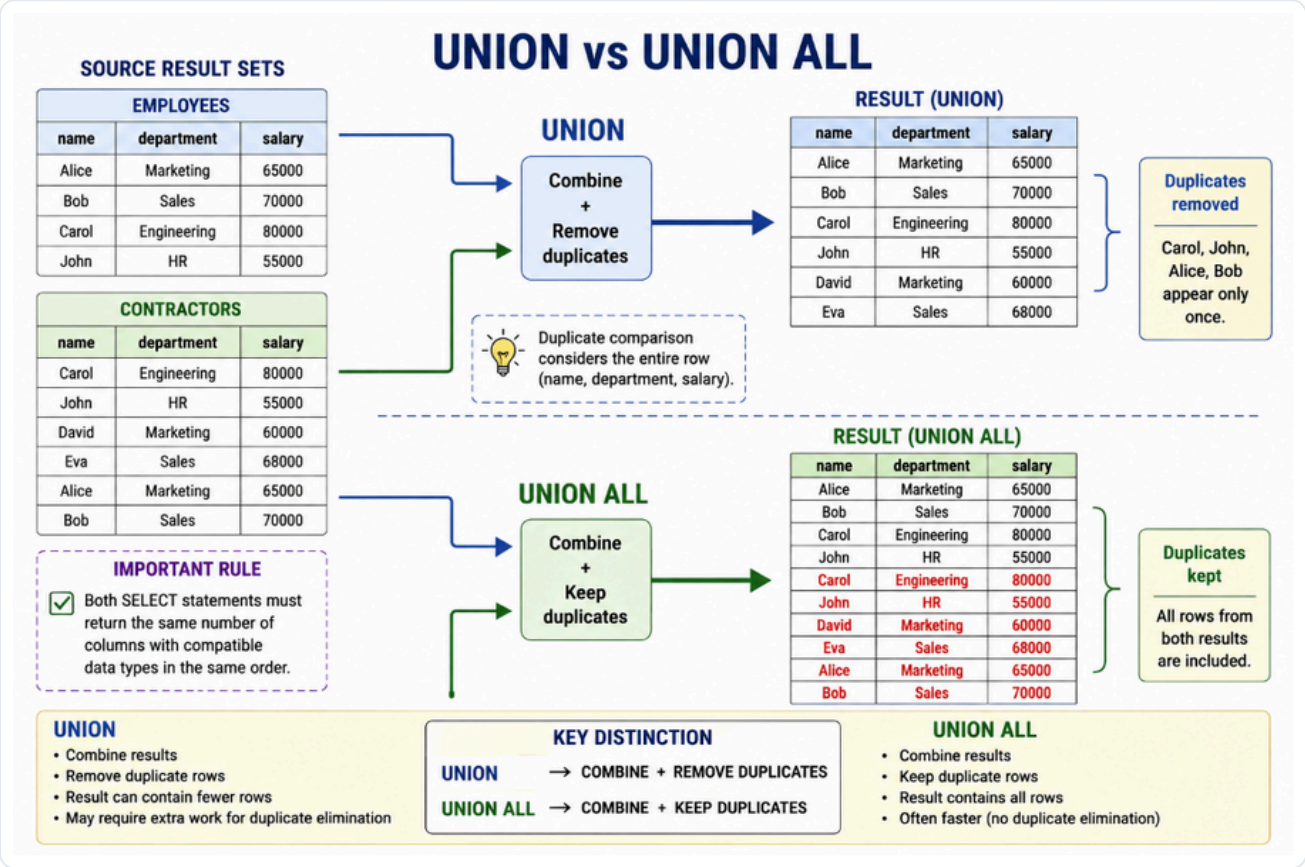


Figure 1: UNION combines result sets and removes duplicate rows, while UNION ALL combines result sets without removing duplicates.

Practice Tables

This topic uses two tables:

Employees

name	department	salary
Alice	Marketing	65000
Bob	Sales	70000
Carol	Engineering	80000
John	HR	55000

Contractors

name	department	salary
David	Marketing	60000
Carol	Engineering	80000
John	HR	55000
Alice	Marketing	65000
Eva	Sales	68000
Bob	Sales	70000

UNION

`UNION` combines the results of two `SELECT` statements and removes duplicate rows.

```sql
SELECT name, department, salary
FROM employees

UNION

SELECT name, department, salary
FROM contractors;
```

The identical rows for `Alice`, `Bob`, `Carol`, and `John` are returned only once.

> ***UNION → Duplicates removed***

UNION ALL

`UNION ALL` combines the results and keeps every row.

```sql
SELECT name, department, salary
FROM employees

UNION ALL

SELECT name, department, salary
FROM contractors;
```

The rows that appear in both tables remain duplicated in the result.

> ***UNION ALL → Duplicates kept***

Important Rule

The `SELECT` statements used with `UNION` or `UNION ALL` must return the **same number of columns**, with compatible data types in corresponding positions.

For example:

```sql
SELECT name, salary
FROM employees

UNION

SELECT name, salary
FROM contractors;
```

Copy

is valid because both queries return two compatible columns.

UNION vs UNION ALL

`UNION`	`UNION ALL`
Removes duplicate rows	Keeps duplicate rows
Usually requires duplicate elimination work	Usually avoids duplicate-removal work
Result can contain fewer rows	Result contains all rows from both results

> *Use UNION when duplicates should be removed. Use UNION ALL when every row should be preserved.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
UNION combines tables permanently	It combines query result sets
UNION ALL removes duplicates	It keeps duplicates
The SELECTs can return different numbers of columns	They must return the same number of compatible columns
UNION compares only the first column	Duplicate elimination considers the entire selected row
UNION ALL is always slower	It avoids duplicate-removal work and can often be more efficient

Placement Quick Points

```text
UNION
→ COMBINE + REMOVE DUPLICATES

UNION ALL
→ COMBINE + KEEP DUPLICATES
```

- `UNION` combines result sets and removes duplicate rows.
- `UNION ALL` combines result sets and keeps duplicates.
- Corresponding `SELECT` statements must return the same number of compatible columns.

- Duplicate comparison considers the complete result row.
- `UNION ALL` is commonly preferred when duplicate removal is not required.

Interview Questions

Q1. What is UNION?

`UNION` combines the results of multiple `SELECT` queries and removes duplicate rows.

Q2. What is UNION ALL?

`UNION ALL` combines the results and keeps duplicate rows.

Q3. What is the main difference?

```
text UNION → Remove duplicates UNION ALL → Keep duplicates
```

Q4. What requirement must SELECT statements satisfy?

They must return the same number of columns, with compatible data types in corresponding positions.

Q5. Which is generally faster, UNION or UNION ALL?

`UNION ALL` is often faster because it does not need to perform duplicate elimination.

Q6. Does UNION combine tables?

No. It combines the **results of SELECT queries** into one result set.

Practice & Hands-On Exercises

Using `employees` and `contractors`:

1. Write a `UNION` query:

```
CODE

SELECT name, department, salary FROM employees
UNION
SELECT name, department, salary FROM contractors;
```

1. Write a `UNION ALL` query:

```
CODE

SELECT name, department, salary FROM employees
UNION ALL
SELECT name, department, salary FROM contractors;
```

1. Compare the number of rows returned.
2. Identify which rows are duplicates.
3. Explain why `UNION ALL` can be more efficient.
4. Write two compatible `SELECT` queries and combine them using `UNION`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE

UNION
→ COMBINE
→ REMOVE DUPLICATES

UNION ALL
→ COMBINE
→ KEEP DUPLICATES
```

> ***UNION combines result sets and removes duplicates, while UNION ALL combines result sets without removing duplicates.***

End of UNION / UNION ALL

11.4 INTERSECT / EXCEPT

Level: 11 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`INTERSECT` and `EXCEPT` compare the results of two `SELECT` queries.

Operator	Purpose
`INTERSECT`	Returns rows present in both results
`EXCEPT`	Returns rows from the first result that are not in the second

> ***INTERSECT → Common rows EXCEPT → First result only***

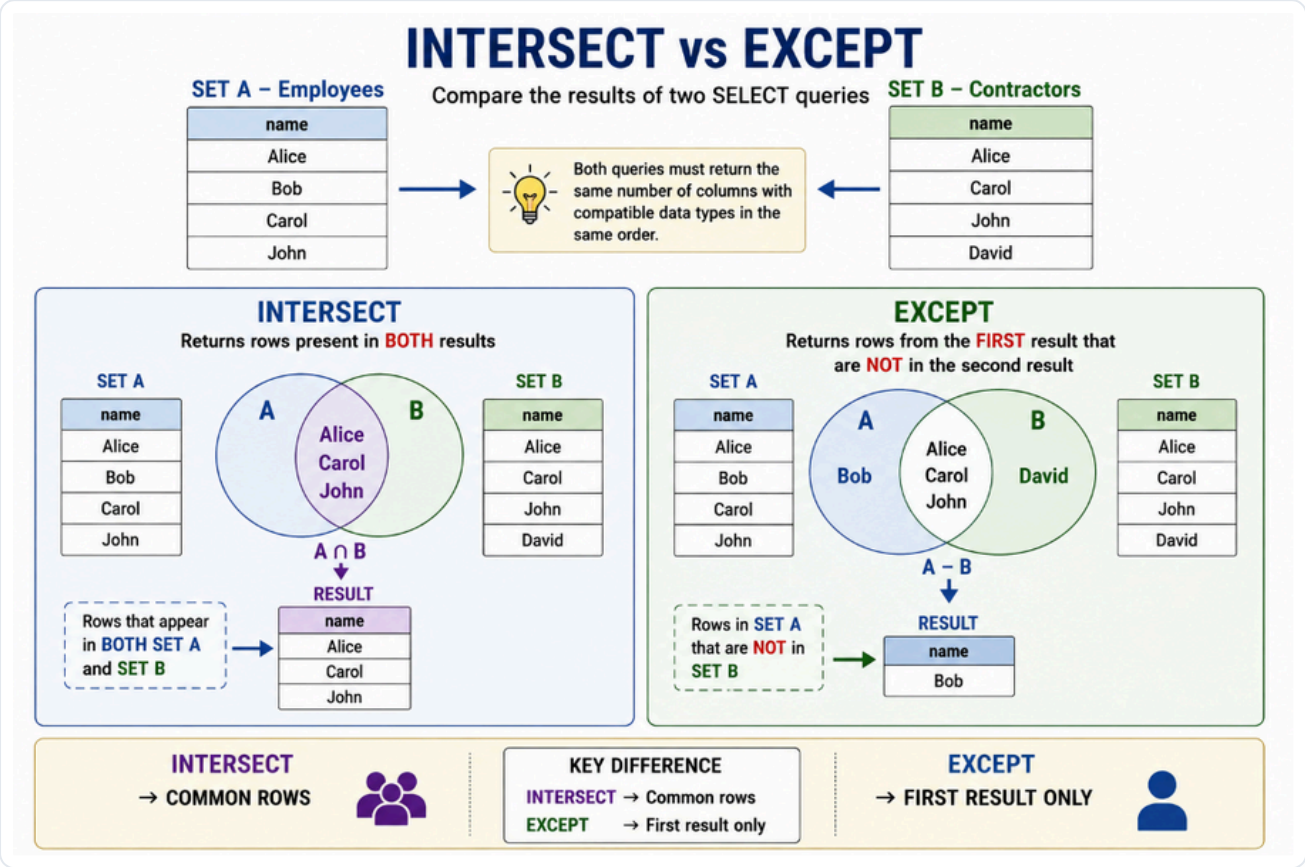


Figure 1: INTERSECT returns common rows, while EXCEPT returns rows present in the first result but not the second.

Practice Tables

This topic reuses the `employees` and `contractors` tables from **11.3**.

INTERSECT

`INTERSECT` returns rows that appear in **both** query results.

```sql
SELECT name, department, salary
FROM employees

INTERSECT

SELECT name, department, salary
FROM contractors;
```

The result contains rows that match in **all selected columns**.

> ***INTERSECT → Common rows***

EXCEPT

`EXCEPT` returns rows from the **first** `SELECT` **result** that do not appear in the second.

```sql
SELECT name, department, salary
FROM employees

EXCEPT

SELECT name, department, salary
FROM contractors;
```

With the original practice data, this returns no rows because every employee row also appears in `contractors`.

> ***EXCEPT → First result minus second result***

Simple Comparison

```
TEXT

INTERSECT
A ∩ B
→ COMMON ROWS

EXCEPT
A − B
→ FIRST RESULT ONLY
```

For example:

```
TEXT

A = Alice, Bob, Carol
B = Alice, Carol, David

INTERSECT → Alice, Carol
EXCEPT → Bob
```

Important Rule

As with `UNION`, the two `SELECT` statements must return the **same number of columns**, with compatible data types in corresponding positions.

```sql
SELECT name, salary FROM employees

INTERSECT

SELECT name, salary FROM contractors;
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`INTERSECT` returns all rows from both queries	It returns only common rows
`EXCEPT` compares tables permanently	It compares query result sets
`EXCEPT` returns rows unique to both sides	It returns rows from the first result that are absent from the second
The SELECTs can have different columns	They must have the same number of compatible columns
Column order does not matter	Corresponding columns must be compatible

Placement Quick Points

```text
INTERSECT
→ COMMON RESULTS

EXCEPT
→ FIRST RESULT ONLY
```

- `INTERSECT` returns common rows.
- `EXCEPT` returns rows from the first result that are not in the second.
- Both operate on `SELECT` result sets.
- Corresponding queries must return the same number of compatible columns.

- These operators are useful for comparing datasets.

Interview Questions

Q1. What is INTERSECT?

`INTERSECT` returns rows that exist in both `SELECT` results.

Q2. What is EXCEPT?

`EXCEPT` returns rows from the first `SELECT` result that do not exist in the second.

Q3. What is the difference?

```
text INTERSECT → Common rows EXCEPT → First result only
```

Q4. What requirement applies to both queries?

They must return the same number of columns with compatible data types in corresponding positions.

Q5. How can EXCEPT be useful?

It can help identify records that exist in one dataset but not another.

Practice & Hands-On Exercises

Using `employees` and `contractors`:

1. Find rows common to both tables using `INTERSECT`:

```
CODE
Copy

SELECT name, department, salary FROM employees
INTERSECT
SELECT name, department, salary FROM contractors;
```

1. Find employee rows that are not present in `contractors` using `EXCEPT`:

```
CODE
Copy

SELECT name, department, salary FROM employees
EXCEPT
SELECT name, department, salary FROM contractors;
```

1. Explain the difference between `INTERSECT` and `EXCEPT`.
2. Identify how these operators can help compare two datasets.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE
Copy

INTERSECT → COMMON
EXCEPT    → FIRST ONLY
```

> ***INTERSECT finds common rows between two query results, while EXCEPT finds rows that exist only in the first result.***

End of INTERSECT / EXCEPT

11.5 Subqueries (WHERE, FROM, SELECT)

Level: 11 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **subquery** is a query written inside another SQL query.

It is useful when one query needs the result of another query.

```text
INNER QUERY
    ↓
RESULT
    ↓
OUTER QUERY
```

Copy

Subqueries can commonly be used in:

- `WHERE`
- `FROM`
- `SELECT`

> *Subquery → A query inside another query.*

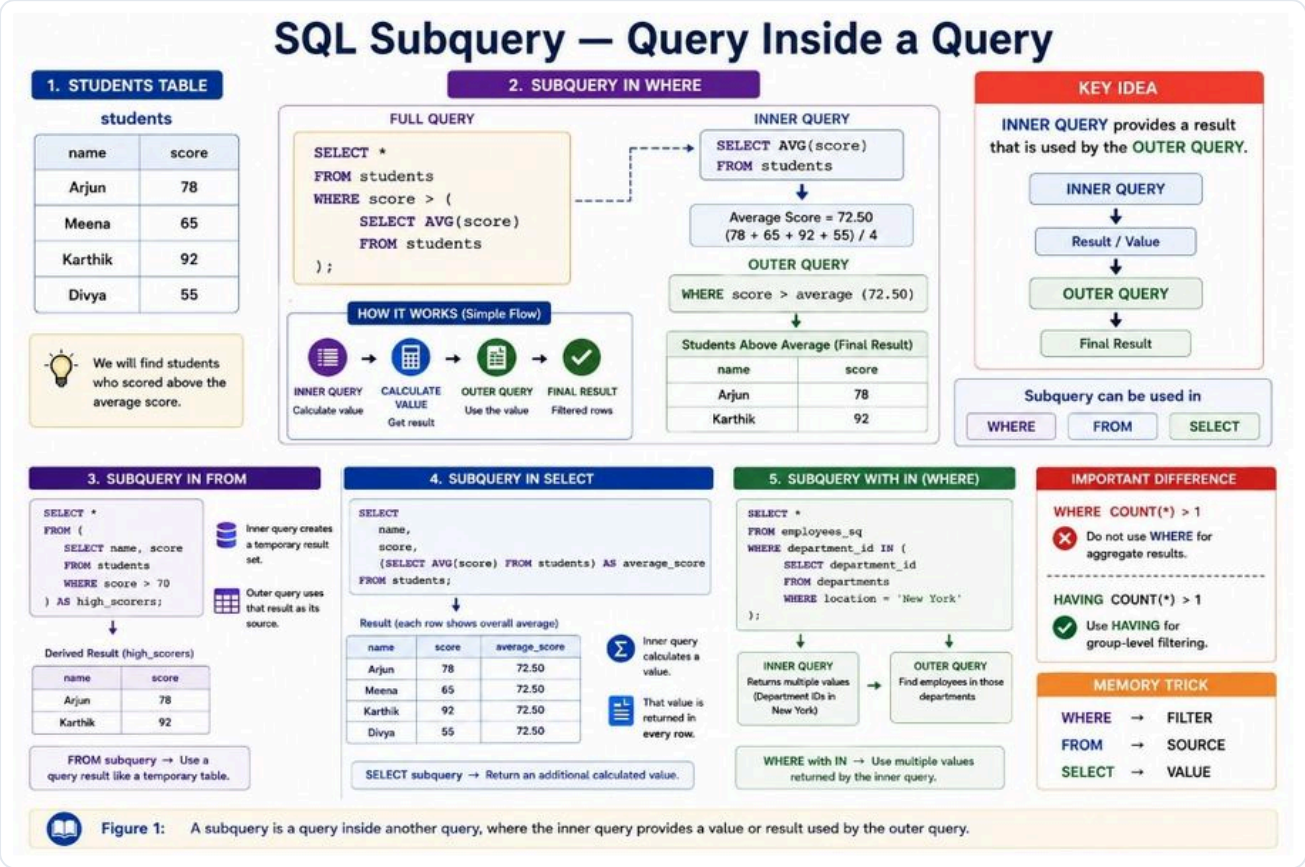


Figure 1: A subquery is a query inside another query, where the inner query provides a value or result used by the outer query.

Subquery in WHERE

A subquery can provide a value used to filter rows.

Find students who scored above the average:

```sql
SELECT *
FROM students
WHERE score > (
    SELECT AVG(score)
    FROM students
);
```

The inner query calculates the average score. The outer query then finds students whose score is greater than that value.

> *WHERE subquery → Use another query's result for filtering.*

Subquery with IN

A subquery can also return multiple values.

Find employees working in departments located in New York:

```sql
SELECT *
FROM employees_sq
WHERE department_id IN (
    SELECT department_id
    FROM departments
    WHERE location = 'New York'
);
```

The inner query returns department IDs, and the outer query finds employees belonging to those departments.

Three Common Places

Location	Purpose
`WHERE`	Filter using another query's result
`FROM`	Use a query result as a temporary result set
`SELECT`	Return a calculated value

Simple Memory Trick

```text
WHERE  → FILTER
FROM   → SOURCE
SELECT → VALUE
```

Subquery in FROM

A subquery in `FROM` can act as a temporary result set.

Example:

```sql
SELECT *
FROM (
    SELECT name, score
    FROM students
    WHERE score > 70
) AS high_scorers;
```

The inner query first creates the `high_scorers` result, which the outer query uses as its source.

> *FROM subquery → Use a query result like a temporary table.*

Subquery in SELECT

A subquery can also return a calculated value in the `SELECT` list.

Example:

```sql
SELECT
    name,
    score,
    (SELECT AVG(score) FROM students) AS average_score
FROM students;
```

This shows each student's score along with the overall average.

> *SELECT subquery → Return an additional calculated value.*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
A subquery is a separate database	It is a query nested inside another query
Every subquery returns multiple rows	A subquery can return one value, one column, or multiple rows depending on the query
A subquery can only be used in WHERE	It can also be used in places such as FROM and SELECT
The inner query always runs completely before the outer query	The optimizer may choose a different physical execution strategy
A subquery always performs worse than a JOIN	Performance depends on the query, database, data, and execution plan

Placement Quick Points

```text
SUBQUERY
→ QUERY INSIDE QUERY

WHERE
→ FILTER

FROM
→ SOURCE

SELECT
→ CALCULATED VALUE
```

- A subquery is a query nested inside another query.

- Subqueries can return a single value or multiple rows depending on their use.
- `WHERE` subqueries are commonly used for filtering.
- `FROM` subqueries provide a derived result set.
- `SELECT` subqueries can provide calculated values.
- Subqueries and joins can sometimes solve similar problems.

Interview Questions

Q1. What is a subquery?

A subquery is a SQL query nested inside another query.

Q2. Where can subqueries be used?

Common locations include:

`text WHERE FROM SELECT`

Q3. What is a WHERE subquery?

A subquery whose result is used to filter rows in the outer query.

Q4. What is a FROM subquery?

A subquery used as a derived result set in the `FROM` clause.

Q5. What is a SELECT subquery?

A subquery used to produce a value or calculated result in the `SELECT` list.

Q6. Can a subquery return multiple rows?

Yes. The allowed result depends on how the subquery is used. For example, `IN` can work with multiple returned values.

Practice & Hands-On Exercises

Using the practice tables:

1. Find students who scored above the average:

```text
SELECT * FROM students
WHERE score > (SELECT AVG(score) FROM students);
```

1. Find employees working in New York departments using `IN`:

```sql
SELECT * FROM employees_sq
WHERE department_id IN (
    SELECT department_id FROM departments WHERE location = 'New York'
);
```

1. Create a `FROM` subquery that returns students scoring above `70`:

```sql
SELECT * FROM (
    SELECT name, score FROM students WHERE score > 70
) AS high_scorers;
```

1. Use a `SELECT` subquery to display each student's score and the overall average:

```sql
SELECT name, score, (SELECT AVG(score) FROM students) AS average_score
FROM students;
```

1. Explain the difference between subqueries in `WHERE`, `FROM`, and `SELECT`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```sql
SUBQUERY
    ↓
PROVIDES A RESULT
    ↓
OUTER QUERY USES IT
```

> *A subquery is a query inside another query, commonly used in WHERE for filtering, FROM for derived results, and SELECT for calculated values.*

End of Subqueries (WHERE, FROM, SELECT)

12.1 Window Functions Overview (OVER, PARTITION BY, ORDER BY)

Level: 12 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

A **window function** performs a calculation across related rows while **keeping every original row** in the result.

This is the key difference from `GROUP BY`:

```text
GROUP BY
→ Combines rows
→ Fewer result rows

WINDOW FUNCTION
→ Calculates across rows
→ Keeps every result row
```

> *Window Function → Calculate across rows without collapsing them.*

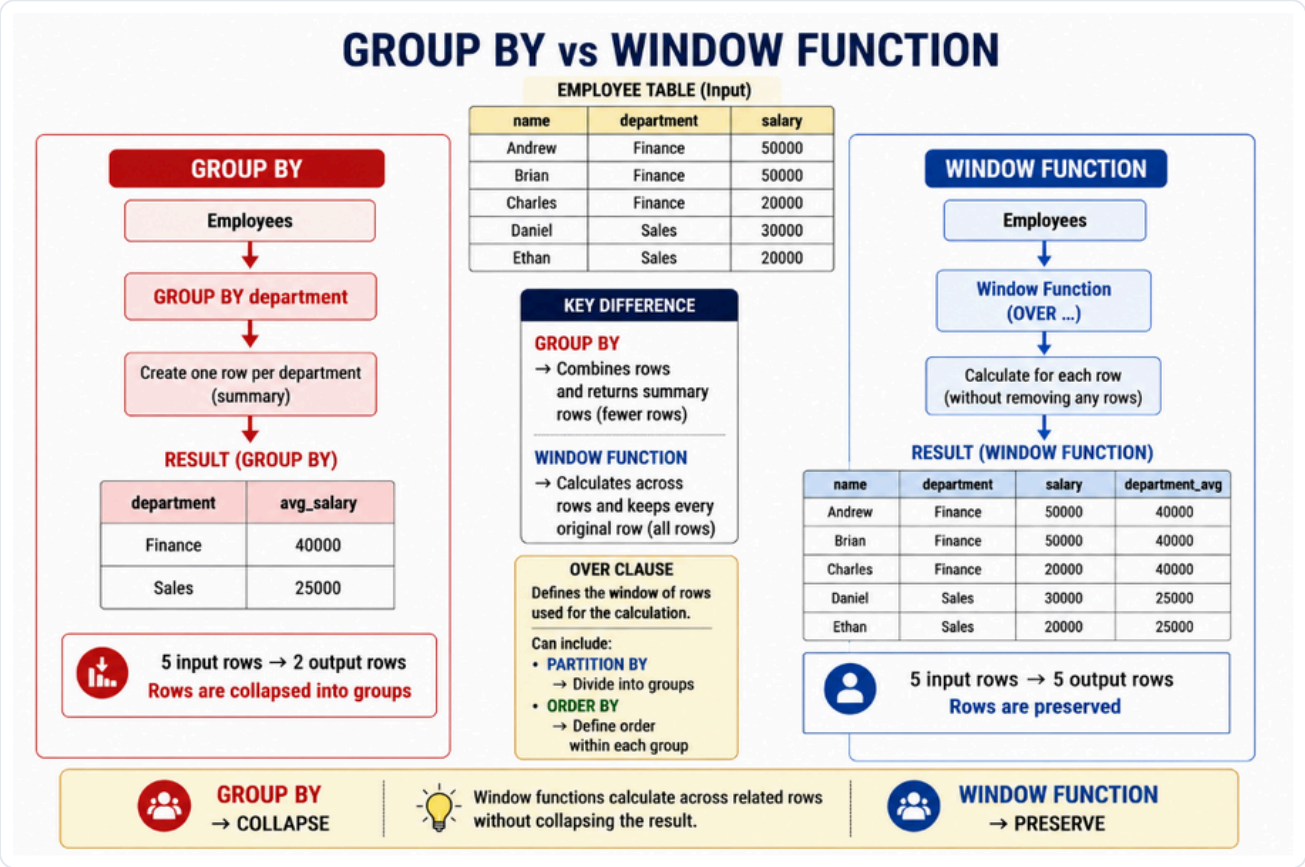


Figure 1: GROUP BY collapses rows into summaries, while window functions calculate across related rows without removing the individual rows.

The OVER Clause

The `OVER` clause defines the **window of rows** used for the calculation.

It can include:

Part	Purpose
`PARTITION BY`	Divides rows into groups
`ORDER BY`	Defines the order within each group

Basic structure:

```SQL
function_name()
OVER (
    PARTITION BY column
    ORDER BY column
)
```

> *OVER → Defines which rows the window function works across.*

PARTITION BY

`PARTITION BY` divides rows into groups **without collapsing them**.

For example:

```SQL
AVG(salary) OVER (
    PARTITION BY department
)
```

This calculates the average salary separately for each department while keeping every employee row.

Conceptually:

```TEXT
Finance
→ Finance average

Sales
→ Sales average
```

ORDER BY

Inside a window function, `ORDER BY` defines the order in which rows are considered.

For example:

```sql
SUM(salary) OVER (
    ORDER BY salary
)
```

The ordering is important for calculations such as **running totals** and rankings.

> *PARTITION BY → Which group? ORDER BY → In what order?*

Simple PostgreSQL Example

Practice table:

```sql
CREATE TABLE employee (
    name VARCHAR(50),
    department VARCHAR(50),
    salary NUMERIC(10,2)
);

INSERT INTO employee (name, department, salary)
VALUES
('Andrew', 'Finance', 50000),
('Brian', 'Finance', 50000),
('Charles', 'Finance', 20000),
('Daniel', 'Sales', 30000),
('Ethan', 'Sales', 20000);
```

Now calculate the average salary for each department:

```sql
SELECT
    name,
    department,
    salary,
    AVG(salary) OVER (
        PARTITION BY department
    ) AS department_avg
FROM employee;
```

Result idea:

name	department	salary	department_avg
Andrew	Finance	50000	40000
Brian	Finance	50000	40000
Charles	Finance	20000	40000
Daniel	Sales	30000	25000
Ethan	Sales	20000	25000

Notice that **all five employees remain in the result**.

GROUP BY vs Window Function

GROUP BY

```SQL
SELECT department, AVG(salary)
FROM employee
GROUP BY department;
```

Result:

```TEXT
Finance → 40000
Sales   → 25000
```

Only one row is returned for each department.

Window Function

```SQL
SELECT
    name,
    department,
    salary,
    AVG(salary) OVER (PARTITION BY department)
FROM employee;
```

Every employee remains visible, with the department average added.

> *GROUP BY → Summary rows Window Function → Original rows + calculated value*

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Window functions remove rows like GROUP BY	Window functions preserve the rows
`PARTITION BY` creates a permanent table	It only defines groups for the calculation
`ORDER BY` inside OVER sorts the final result automatically	It defines the window calculation order; final output ordering is a separate concern
Every window function needs `PARTITION BY`	`PARTITION BY` is optional
Every window function needs `ORDER BY`	`ORDER BY` is optional for some window calculations

Placement Quick Points

```text
OVER
→ DEFINES WINDOW

PARTITION BY
→ DIVIDE INTO GROUPS

ORDER BY
→ DEFINE ORDER

WINDOW FUNCTION
→ KEEP ROWS
```

- Window functions calculate across related rows.
- They preserve individual result rows.
- `OVER` defines the window.
- `PARTITION BY` divides rows into groups without collapsing them.
- `ORDER BY` controls the order within the window.
- Window functions are useful for rankings, running totals, comparisons, and other row-by-row analytics.

Interview Questions

Q1. What is a window function?

A window function performs a calculation across related rows while keeping the individual rows in the result.

Q2. What is the purpose of OVER?

`OVER` defines the rows used by the window calculation.

Q3. What does PARTITION BY do?

It divides rows into groups for the window calculation without collapsing those rows.

Q4. What does ORDER BY do inside OVER?

It defines the order of rows within the window.

Q5. What is the main difference between GROUP BY and a window function?

```text
GROUP BY → COLLAPSES ROWS

WINDOW FUNCTION → PRESERVES ROWS
```

Q6. Is PARTITION BY mandatory?

> No. It is optional and depends on the calculation.

Practice & Hands-On Exercises

Using the `employee` table:

1. Calculate the average salary for each department using `PARTITION BY`:

```
CODE
Copy

SELECT name, department, salary,
       AVG(salary) OVER (PARTITION BY department) AS department_avg
FROM employee;
```

1. Display each employee along with their department average.
2. Explain why the window-function query returns all employees.
3. Compare the result with a `GROUP BY` query.
4. Try `ORDER BY salary` inside `OVER` and observe the effect:

```
CODE
Copy

SELECT name, department, salary,
       SUM(salary) OVER (PARTITION BY department ORDER BY salary) AS running_total
FROM employee;
```

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

CODE

Copy

```
OVER
 ↓
PARTITION BY → GROUP
 ↓
ORDER BY → ORDER
 ↓
CALCULATE
 ↓
KEEP ALL ROWS
```

Window functions perform calculations across related rows without collapsing the result, with `OVER`, `PARTITION BY`, and `ORDER BY` defining the calculation window.

End of Window Functions Overview (OVER, PARTITION BY, ORDER BY)

12.2 RANK() and DENSE_RANK()

Level: 12 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`RANK()` and `DENSE_RANK()` assign a rank to rows based on an `ORDER BY`.

The main difference is **how they handle ties**.

Function	Tie Handling
`RANK()`	Same rank, then **skips** the next rank
`DENSE_RANK()`	Same rank, but **does not skip**

> *RANK → Tie + Gap DENSE_RANK → Tie + No Gap*

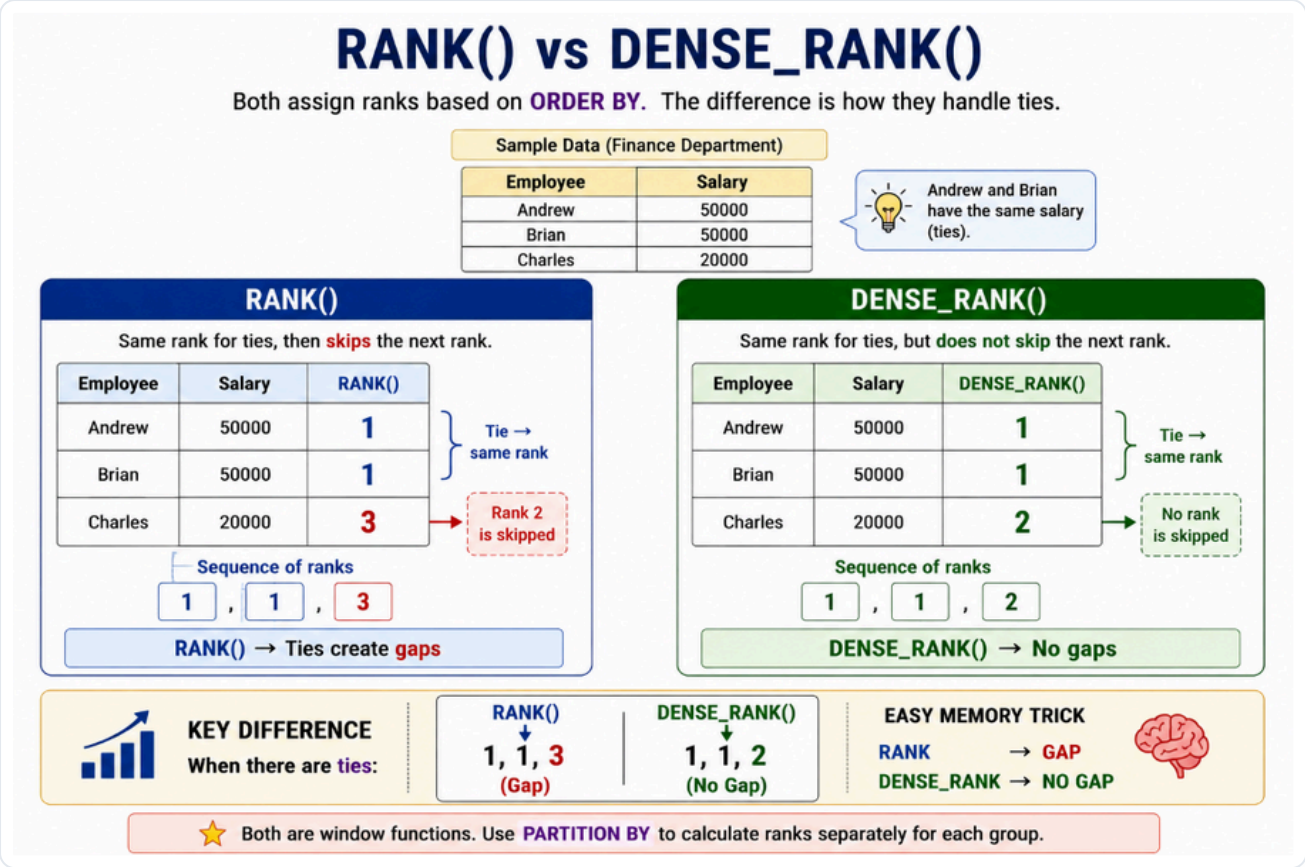


Figure 1: RANK() leaves gaps after ties, while DENSE_RANK() keeps the ranking sequence continuous.

Practice Table

This topic reuses the `employee` table from 12.1.

name	department	salary
Andrew	Finance	50000
Brian	Finance	50000
Charles	Finance	20000
Daniel	Sales	30000
Ethan	Sales	20000

RANK()

`RANK()` gives the same rank to tied rows, but **skips rank numbers after the tie**.

```sql
SELECT
    name,
    department,
    salary,
    RANK() OVER (
        PARTITION BY department
        ORDER BY salary DESC
    ) AS emp_rank
FROM employee;
```

Finance:

name	salary	rank
Andrew	50000	1
Brian	50000	1
Charles	20000	3

Andrew and Brian tie at `1`, so the next rank is `3`.

> *RANK() → Ties create gaps.*

DENSE_RANK()

`DENSE_RANK()` also gives the same rank to tied rows, but **does not skip the next rank**.

```sql
SELECT
    name,
    department,
    salary,
    DENSE_RANK() OVER (
        PARTITION BY department
        ORDER BY salary DESC
    ) AS emp_dense_rank
FROM employee;
```

Finance:

name	salary	dense_rank
Andrew	50000	1
Brian	50000	1
Charles	20000	2

> *DENSE_RANK() → Ties do not create gaps.*

RANK vs DENSE_RANK

```TEXT
Salary: 50000, 50000, 20000

RANK()
→ 1, 1, 3

DENSE_RANK()
→ 1, 1, 2
```

Easy Memory Trick

```TEXT
RANK       → GAP
DENSE_RANK → NO GAP
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`RANK()` gives every row a unique number	Tied rows receive the same rank
`DENSE_RANK()` skips after ties	It does not skip
Both functions always return `1, 2, 3...`	Ties can change the sequence
`PARTITION BY` is required	It is optional
Ranking changes the stored table	It only adds a calculated value to the query result

Placement Quick Points

```text
RANK()
→ SAME RANK FOR TIES
→ GAPS AFTER TIES

DENSE_RANK()
→ SAME RANK FOR TIES
→ NO GAPS
```

- Both functions are window functions.
- Both assign ranks according to the `ORDER BY`.
- `RANK()` leaves gaps after ties.
- `DENSE_RANK()` does not leave gaps.

- `PARTITION BY` can create separate rankings for each group.

Interview Questions

Q1. What is RANK()?

`RANK()` assigns ranks to rows and gives tied rows the same rank, leaving gaps after ties.

Q2. What is DENSE_RANK()?

`DENSE_RANK()` assigns the same rank to tied rows without leaving gaps.

Q3. What is the main difference?

```
text RANK → 1, 1, 3 DENSE_RANK → 1, 1, 2
```

Q4. Can rankings be calculated separately by department?

Yes, using:

```
Sql PARTITION BY department
```

Q5. Does RANK() modify the table?

No. It calculates the rank in the query result.

Practice & Hands-On Exercises

Using the `employee` table:

1. Rank employees by salary within each department.
2. Apply `RANK()`:

```
SELECT name, department, salary,
       RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS emp_rank
FROM employee;
```

1. Apply `DENSE_RANK()`:

```
SELECT name, department, salary,
       DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS emp_dense_rank
FROM employee;
```

1. Compare the results when salaries are tied.
2. Explain why `RANK()` produces a gap.
3. Explain why `DENSE_RANK()` does not.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
RANK()
→ 1, 1, 3

DENSE_RANK()
→ 1, 1, 2
```

> ***Both functions handle ties with the same rank, but RANK() leaves gaps while DENSE_RANK() keeps the ranking continuous.***

End of RANK() and DENSE_RANK()

12.3 ROW_NUMBER()

Level: 12 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`ROW_NUMBER()` assigns a **unique number to every row** based on the specified order.

Unlike `RANK()` and `DENSE_RANK()`, **two rows never receive the same row number**.

```text
ROW_NUMBER()
→ UNIQUE NUMBER FOR EVERY ROW
```

> *ROW_NUMBER() → No ties*

Figure 1: ROW_NUMBER() assigns a unique sequential number to every row, even when ordered values are tied.

Practice Table

This topic reuses the `employee` table from 12.1.

name	department	salary
Andrew	Finance	50000
Brian	Finance	50000
Charles	Finance	20000
Daniel	Sales	30000
Ethan	Sales	20000

Example: ROW_NUMBER()

SQL

Copy

```sql
SELECT
    name,
    department,
    salary,
    ROW_NUMBER() OVER (
        PARTITION BY department
        ORDER BY salary DESC
    ) AS row_no
FROM employee;
```

Result:

name	department	salary	row_no
Andrew	Finance	50000	1
Brian	Finance	50000	2
Charles	Finance	20000	3
Daniel	Sales	30000	1
Ethan	Sales	20000	2

Even though Andrew and Brian have the same salary, their row numbers are different.

> ***ROW_NUMBER() → Every row gets a unique number.***

How It Works

```
TEXT

PARTITION BY department
        ↓
Separate Finance and Sales
        ↓
ORDER BY salary DESC
        ↓
Assign row numbers
```

For Finance:

```
TEXT

50000 → 1
50000 → 2
20000 → 3
```

The exact order of tied rows should not be relied upon unless the `ORDER BY` includes an additional tie-breaker.

For example:

```sql
ROW_NUMBER() OVER (
    PARTITION BY department
    ORDER BY salary DESC, name
)
```

This gives the database another column to determine the order of tied salaries.

RANK vs DENSE_RANK vs ROW_NUMBER

Salary	RANK()	DENSE_RANK()	ROW_NUMBER()
50000	1	1	1
50000	1	1	2
20000	3	2	3

```text
RANK       → 1, 1, 3
DENSE_RANK → 1, 1, 2
ROW_NUMBER → 1, 2, 3
```

> *ROW_NUMBER() never assigns the same number to two rows within the window.*

Common Uses

`ROW_NUMBER()` is commonly used for:

- Numbering rows
- Finding the top N rows per group
- Selecting the first/latest record
- Removing duplicate records using a ranking rule

Example:

```sql
SELECT *
FROM (
    SELECT *,
           ROW_NUMBER() OVER (
               PARTITION BY department
               ORDER BY salary DESC
           ) AS row_no
    FROM employee
) t
WHERE row_no = 1;
```

This returns the highest-salary employee from each department.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Equal salaries receive the same row number	Every row gets a unique number
`ROW_NUMBER()` leaves gaps after ties	It produces sequential numbers
`ROW_NUMBER()` changes the table	It only adds a calculated value to the result
Tied rows always have a predictable order	Add a tie-breaker to `ORDER BY` when deterministic ordering is required
`PARTITION BY` is mandatory	It is optional

Placement Quick Points

```text
ROW_NUMBER()
→ UNIQUE NUMBER

TIE
→ NO SHARED NUMBER

PARTITION BY
→ SEPARATE GROUPS

ORDER BY
→ DEFINE NUMBERING ORDER
```

- `ROW_NUMBER()` assigns a unique sequential number to each row.
- Tied values still receive different row numbers.

- `PARTITION BY` can restart numbering for each group.
- `ORDER BY` determines the numbering order.
- Add a tie-breaker when a deterministic order is required.

Interview Questions

Q1. What is ROW_NUMBER()?

`ROW_NUMBER()` assigns a unique sequential number to each row within the window.

Q2. How does ROW_NUMBER() handle ties?

Tied values still receive different row numbers.

Q3. What is the difference between RANK() and ROW_NUMBER()?

```
text RANK → Ties share a rank ROW_NUMBER → Every row gets a unique number
```

Q4. Can ROW_NUMBER() use PARTITION BY?

Yes. It can restart numbering for each partition.

Q5. Why add another column to ORDER BY?

To provide a tie-breaker and make the row-number assignment deterministic when primary ordering values are equal.

Practice & Hands-On Exercises

Using the `employee` table:

1. Assign row numbers to all employees by salary descending:

```
CODE
Copy

SELECT name, department, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS overall_row_no
FROM employee;
```

1. Assign row numbers separately for each department:

```
CODE
Copy

SELECT name, department, salary,
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS dept_row_no
FROM employee;
```

1. Observe how tied salaries are numbered.
2. Compare `ROW_NUMBER()` with `RANK()`.
3. Add a second `ORDER BY` column and observe the effect:

```
CODE
Copy

SELECT name, department, salary,
       ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC, name ASC) AS tie_broken_row
FROM employee;
```

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
CODE
Copy

ROW_NUMBER()
      ↓
UNIQUE SEQUENTIAL NUMBER
      ↓
NO TIES
```

> *ROW_NUMBER() assigns a unique number to every row, even when multiple rows have the same ordering value.*

End of ROW_NUMBER()

12.4 LAG()

Level: 12 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

`LAG()` returns a value from a **previous row** in the window.

It is useful when you want to compare the current row with an earlier row.

> ***LAG() → Look backward***

Example uses:

```
TEXT

Current Revenue
      ↓
Previous Revenue
      ↓
Compare
```

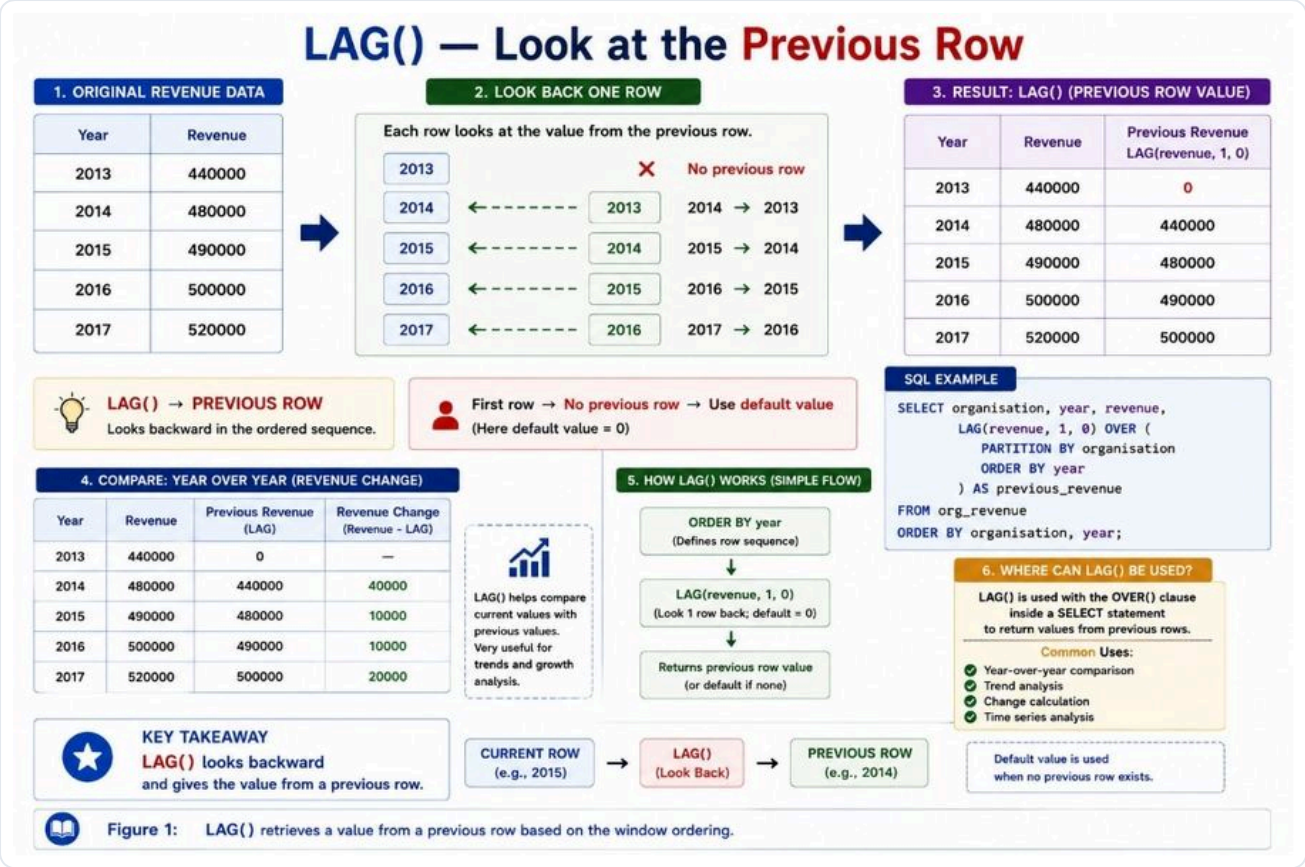


Figure 1: LAG() retrieves a value from a previous row based on the window ordering.

Syntax

```sql
LAG(column_name, offset, default_value)
```

Parameter	Meaning
`column_name`	Value to retrieve
`offset`	Rows to look back; default is `1`
`default_value`	Value used when no previous row exists

Practice Table

This topic uses:

```sql
CREATE TABLE org_revenue (
    organisation VARCHAR(50),
    year INT,
    revenue NUMERIC(12,2)
);

INSERT INTO org_revenue
(organisation, year, revenue)
VALUES
('ABCD News', 2013, 440000),
('ABCD News', 2014, 480000),
('ABCD News', 2015, 490000),
('ABCD News', 2016, 500000),
('ABCD News', 2017, 520000);
```

Example: LAG()

```sql
SELECT
    organisation,
    year,
    revenue,
    LAG(revenue, 1, 0) OVER (
        PARTITION BY organisation
        ORDER BY year
    ) AS previous_revenue
FROM org_revenue
ORDER BY organisation, year;
```

Result:

year	revenue	previous_revenue
2013	440000	0
2014	480000	440000
2015	490000	480000
2016	500000	490000
2017	520000	500000

For `2013`, there is no previous row, so the specified default value `0` is returned.

How LAG() Works

```text
2013 → no previous row → 0
2014 → previous = 2013
2015 → previous = 2014
2016 → previous = 2015
2017 → previous = 2016
```

The `ORDER BY year` determines what "previous" means.

`PARTITION BY organisation` makes the calculation restart for each organisation.

> *ORDER BY → Defines the row sequence LAG() → Looks backward in that sequence*

Comparing Current and Previous Values

A common use is calculating the change:

```sql
SELECT
    year,
    revenue,
    revenue - LAG(revenue) OVER (
        ORDER BY year
    ) AS revenue_change
FROM org_revenue;
```

For example:

```text
2014 → 480000 - 440000 = 40000
2015 → 490000 - 480000 = 10000
```

This is why `LAG()` is useful for **year-over-year comparisons**.

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
`LAG()` looks at the next row	It looks at a previous row
The previous row is determined automatically	`ORDER BY` defines the sequence
`LAG()` always returns a value for the first row	The first row has no previous row and returns `NULL` or the specified default
`LAG()` changes table data	It only calculates a value in the query result
`PARTITION BY` is mandatory	It is optional

Placement Quick Points

```text
LAG()
→ PREVIOUS ROW

ORDER BY
→ DEFINE SEQUENCE

OFFSET
→ HOW MANY ROWS BACK

DEFAULT
→ VALUE WHEN NO PREVIOUS ROW
```

Copy

- `LAG()` retrieves a value from a previous row.
- `ORDER BY` determines the row sequence.

- The default offset is `1`.
- A default value can be supplied for the first row.
- `LAG()` is useful for comparing current and previous values.

Interview Questions

Q1. What is LAG()?

`LAG()` is a window function that retrieves a value from a previous row.

Q2. What determines the previous row?

The `ORDER BY` inside the window definition determines the sequence.

Q3. What happens on the first row?

If there is no previous row, `LAG()` returns `NULL` unless a default value is specified.

Q4. What does the offset do?

It specifies how many rows to look backward.

```
text LAG(value, 1) → Previous row LAG(value, 2) → Two rows back
```

Q5. What is a common use of LAG()?

Comparing the current value with a previous value, such as year-over-year revenue.

Practice & Hands-On Exercises

Using `org_revenue`:

1. Display the previous year's revenue:

```
TEXT
Copy

SELECT organisation, year, revenue,
       LAG(revenue, 1, 0) OVER (PARTITION BY organisation ORDER BY year) AS prev_year_revenue
FROM org_revenue;
```

1. Find the revenue change from the previous year:

```
SQL
Copy

SELECT year, revenue,
       revenue - LAG(revenue) OVER (ORDER BY year) AS revenue_change
FROM org_revenue;
```

1. Use `LAG(revenue, 2)` to look two years back.
2. Explain why the first row has no previous value.
3. Explain the role of `ORDER BY`.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```
SQL
Copy

CURRENT ROW
     ↓
   LAG()
     ↓
PREVIOUS ROW
```

> ***LAG() looks backward within the window and retrieves a value from a previous row, making it useful for comparing current and earlier records.***

End of LAG()

12.5 Table Partitioning (RANGE, LIST, HASH)

Level: 12 | **Version:** v1.1 | **Author:** Meptrasoft

Overview

Table partitioning divides a large table into smaller partitions based on column values.

This can improve query performance because the database may read only the relevant partitions instead of the entire table.

```text
TEXT

LARGE TABLE
    ↓
PARTITIONS
    ↓
SMALLER DATA CHUNKS
```

Copy

> *Partitioning → Divide a large table into smaller parts.*

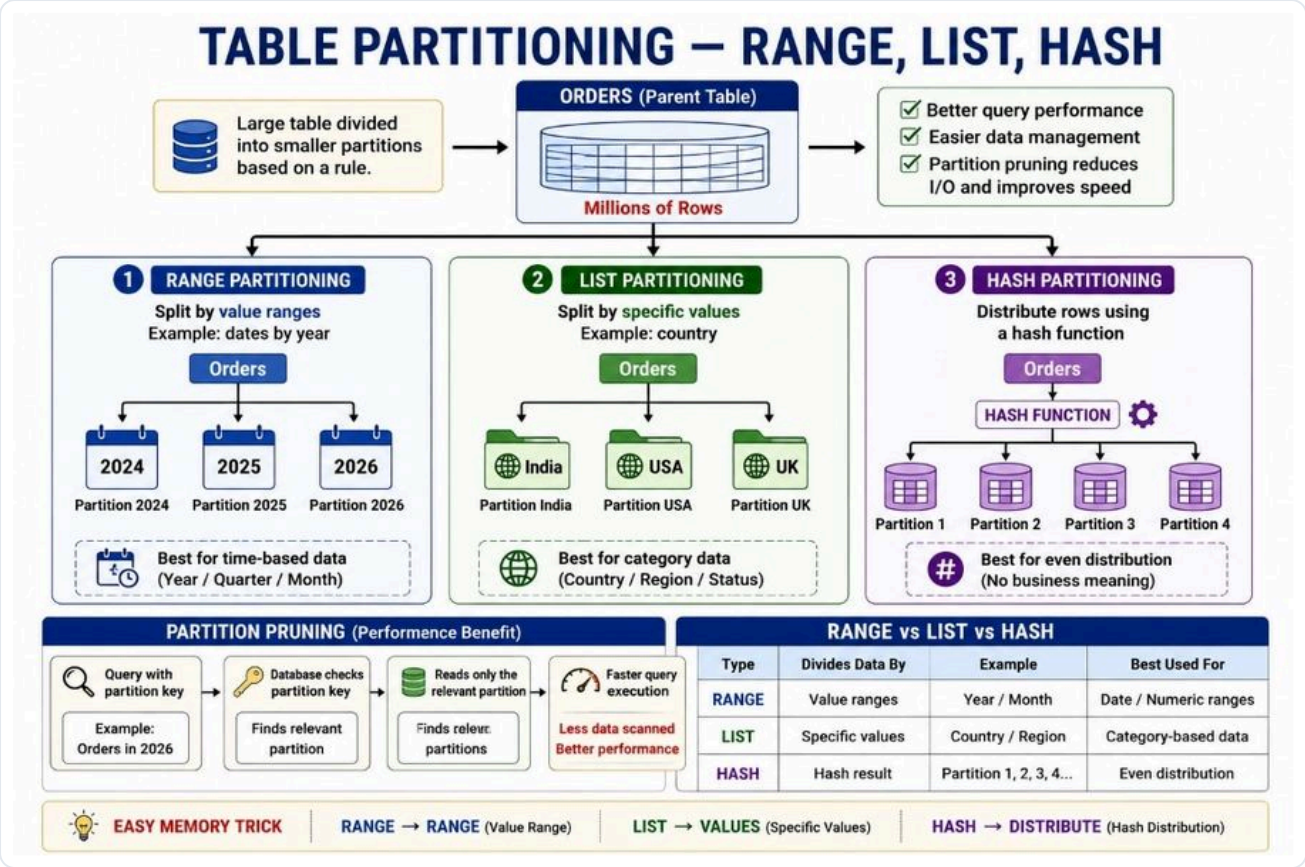


Figure 1: Table partitioning divides a large table into smaller partitions using range, list, or hash rules.

RANGE Partitioning

RANGE partitioning divides rows according to a range of values.

It is commonly used for **dates**.

Example:

Conceptually:

```
TEXT

2024 → Partition 2024
2025 → Partition 2025
2026 → Partition 2026
```

> *RANGE → Divide by value ranges.*

LIST Partitioning

LIST partitioning divides rows based on specific values.

For example, data can be divided by country:

```
TEXT

India → Partition 1
USA   → Partition 2
UK    → Partition 3
```

This is useful when the partitioning column contains a known set of categories.

> *LIST → Divide by specific values.*

HASH Partitioning

```text
Rows
 ↓
HASH FUNCTION
 ↓
Partition 1
Partition 2
Partition 3
Partition 4
```

Unlike range or list partitioning, the rows are distributed based on the hash result rather than a meaningful business category.

> ***HASH → Distribute rows across partitions.***

Partition Pruning

One important benefit of partitioning is **partition pruning**.

Suppose an orders table is partitioned by year and the query asks for 2026 orders:

```sql
SELECT *
FROM orders
WHERE order_date >= '2026-01-01'
  AND order_date < '2027-01-01';
```

The database may be able to access only the relevant 2026 partition rather than scanning every partition.

```text
Query
 ↓
Partition Key
 ↓
Relevant Partition
 ↓
Read Data
```

> *Partition pruning → Skip partitions that cannot contain the requested rows.*

RANGE vs LIST vs HASH

Type	Divides Data By	Example
RANGE	Value ranges	Year / Month
LIST	Specific values	Country / Region
HASH	Hash result	Even distribution

Easy Memory Trick

```text
RANGE → RANGE
LIST  → VALUES
HASH  → DISTRIBUTE
```

Common Beginner Mistakes

✗ Mistake	✓ Correct Understanding
Partitioning creates unrelated tables	Partitions belong to a logical parent table
RANGE is always better than HASH	The suitable strategy depends on the workload
Partitioning automatically makes every query faster	Queries benefit mainly when the partitioning strategy matches the workload
HASH partitions have business meaning	Hash partitioning is primarily for distribution
More partitions always means better performance	Too many partitions can increase planning and management overhead

Placement Quick Points

```text
RANGE
→ VALUE RANGES

LIST
→ SPECIFIC VALUES

HASH
→ HASH DISTRIBUTION
```

- Partitioning divides a large table into smaller partitions.
- **RANGE** is commonly used for dates and numeric ranges.

- **LIST** is useful for categories such as country or region.
- **HASH** distributes rows using a hash function.
- Partition pruning can reduce the amount of data scanned.
- Partitioning should be designed based on query and data-access patterns.

Interview Questions

Q1. What is table partitioning?

> Partitioning divides a large logical table into smaller partitions based on a partitioning rule.

Q2. What is RANGE partitioning?

> It divides rows according to value ranges, commonly dates.

Q3. What is LIST partitioning?

> It divides rows according to specific values or categories.

Q4. What is HASH partitioning?

> It distributes rows across partitions using a hash function.

Q5. What is partition pruning?

> Partition pruning is the process of skipping partitions that cannot contain the rows needed by a query.

Q6. Which partitioning type is commonly used for date-based data?

> RANGE partitioning.

Practice & Hands-On Exercises

1. Identify a suitable partitioning method for sales data by year.
2. Identify a suitable method for country-based data.
3. Explain how HASH partitioning distributes rows.
4. Explain partition pruning in your own words.
5. Compare RANGE, LIST, and HASH.

💡 *Tip: Test your queries using the in-browser interactive runner above.*

Key Takeaway

```text
RANGE → VALUE RANGES
LIST  → SPECIFIC VALUES
HASH  → DISTRIBUTION
```

> ***Table partitioning divides large tables into smaller partitions, and the right partitioning strategy can reduce the amount of data a query needs to scan.***

End of Table Partitioning (RANGE, LIST, HASH)