Relational Algebra Overview
- *Relational Algebra (RA)*: Formal language for querying/manipulating relational databases, underpinning SQL.
- Purpose: Translates high-level queries into executable operations; optimizes execution.

Basic Operations
- *Union (∪)*: Combines tuples, removes duplicates. Formula: (R1 ∪ R2)
Example: ActiveEmployees ∪ RetiredEmployees → unique employees.
⚠️ Same schema required.

-*Set Difference (−)*: Tuples in (R1) not in (R2).
Formula: (R1 - R2)
Example: AllEmployees - RetiredEmployees → current employees.
⚠️ Same schema required.

- *Selection (σ)*: Filters tuples by condition (c).
Formula: (σ_c(R))
Example: σ_{Salary > 40000}(Employee) → employees earning > $40K.
Efficiency: O(N) scan; O(log N) with index.

- *Projection (π)*: Selects columns, removes duplicates.
Formula: (π_{A1,...,An}(R))
Example: π_{SSN, Name}(Employee) → unique SSN, Name pairs.
Efficiency: O(N) hashing; O(N log N) sorting.

- *Cartesian Product (×)*: Combines all tuples from (R1) and (R2).
Formula: (R1 × R2)
Example: Employee × Dependents → all pairs.
Efficiency: O(M × N) I/Os.

- *Renaming (ρ)*: Changes attribute names.
Formula: (ρ_{B1,...,Bn}(R))
Example: ρ_{LastName, SocSecNo}(Employee) → renamed attributes.
Efficiency: No I/O.

Derived Operations

- *Intersection (∩)*: Common tuples in both relations.
Formula: (R1 ∩ R2 = R1 - (R1 - R2))
Example: UnionizedEmployees ∩ RetiredEmployees → common employees.
Efficiency: M + N reads + P writes.
Implementation:
1. Sort Merge intersection
2. Hash Base intersection(Cost you'll find in the actual sort method)

- *Theta Join (⋈_θ)*: Combines tuples where condition (θ) is true.
Formula: (R1 ⋈_theta R2 = σ _θ (R1 \times R2))
Example: Sells ⋈_{Sells.bar = Bars.name} Bars → matching bar names.
Efficiency: O(M × N) I/Os.

- *Natural Join (⋈)*: Joins on common attributes.
Formula: (R1 ⋈ R2)
Example: Employee ⋈ Dependents → matches on shared attributes.
Efficiency: M + N reads + P writes (hash join).

- *Equi-Join*: Theta join with equality condition.
Formula: (R1 ⋈_{A=B} R2)
Example: R1 ⋈_{A=B} R2 → joins on (A = B).
Efficiency: O(M + N) hash join.

Join Overview

Join: Combines tuples from relations (R) and (S) based on a condition (e.g., (R.X = S.Y)).
Purpose: Links related data across tables (e.g., employees with departments).
Why Output Cost Ignored: Identical output across methods; varies by tuple size; focus on operational I/Os.

Theta Join (⋈_θ)
Theta Join: Combines tuples where condition (**θ**) (e.g., (R.X > S.Y)) is true.
Formula: (R1 ⋈_θ R2 = σ _θ (R1 * R2))
Example: Sells ⋈_{Sells.bar = Bars.name} Bars → matching bar names.
Implementation:
Compute Cartesian Product (R1 * R2).
Apply selection (σ _θ) to filter tuples.
Estimated Cost:
I/Os: **(M +⌈ M / B ⌉* N)** (Cartesian Product cost).
Computational: **O(M * N)** for condition checks.

Natural Join (⋈)
Natural Join: Joins on common attributes with equal values.
Formula: (R1 ⋈ R2)
Example: Employee ⋈ Dependents → matches on shared attributes (e.g., SSN).
Implementation:
Identify common attributes.
Perform Equi-Join on these attributes; output combined tuples with single copy of common attributes.
Estimated Cost:
I/Os: **(M + N)** (hash join); **(M +⌈ M / B ⌉ * N)** (nested loops).
Computational: **O(M + N)** (hash join); **O(M * N)** (nested loops).
⚠️ Same schema for common attributes required.

Equi-Join
Equi-Join: Theta Join with equality condition (e.g., (R.A = S.B)).
Formula: (R1 ⋈_{A=B} R2)
Example: R1 ⋈_{A=B} R2 → joins on (A = B).
Implementation: Similar to Theta Join, but optimize for equality (e.g., use hash table).
Estimated Cost:
I/Os: **(M + N)** (hash join); **(M +⌈ M / B ⌉ * N)** (nested loops).
Computational: **O(M + N)** (hash join).

Database Join Algorithms

Simple Nested loop Join
For each tuple in R we scan through each tuple in S.
Cost: M + (P_R * M * N), where P_R is the number of tuples per page in R.

Page Oriented Nested Loop Join
For each page in R we scan through each page in S
Cost: M + (M * N)

Block Nested Loop Join
Block Nested Loop Join: Processes outer relation in blocks to reduce I/Os.
Formula (Cost): (R +⌈ R / (B - 2) ⌉ * S)
Example: (R=10,000, S=8,000, B=2,001), cost 58,000 I/Os (smaller as outer).
Implementation:
Allocate (B-2) buffers for outer relation block, 1 for inner, 1 for output.
For each block of (R), build hash table; scan (S), probe for matches.
Estimated Cost:
I/Os: (R +⌈ R / (B - 2) ⌉ * S)
Computational: O(R \times S) per block, reduced by hashing.
Optimization: Choose smaller relation as outer to minimize (⌈ R / (B - 2) ⌉)

Sort-Merge Join
Sort-Merge Join: Sorts both relations, merges matching tuples.
Example: (R(sid: 28, 31), S(sid: 28, 31)), outputs pairs like ((28, 28)).
Implementation:
Sort (R) and (S) using external sorting.
Merge: Scan both, output matches when (R.X = S.Y); handle duplicates efficiently.
Estimated Cost:
Sorting: **(4M + 4N)** (2 passes each).
Merging: **(M + N)**; up to (M * N) with many duplicates.
Total: **(5M + 5N)** (typical); **(4M + 4N + M * N)** (worst).
Key Point: Efficient if pre-sorted; sensitive to duplicates.
⚠️ Common mistake: Using in-memory sort cost ((M log M)) instead of external ((4M)).

Hash Join
Hash Join: Partitions relations into buckets, joins matching buckets.
Formula (Cost): (3M + 3N)
Example: (R(sid: 1, 4, 7)), hashed by sid mod 3, joins with (S).
Implementation:
Partition (R) and (S) into buckets using hash function (h).
For each bucket pair, build hash table for (R)-bucket, probe with (S)-bucket.
Estimated Cost:
Partitioning: **(2M + 2N)** (read/write each relation).
Probing: **(M + N)** (read buckets).
Total: **(3M + 3N)**.
Assumption: (R)-partitions fit in memory ((M / (B-1) \leq B-2)).
⚠️ Common mistake: Applying to non-equi joins (requires equality).

Index Nested Loop Join
Index Nested Loop Join: Uses index on inner relation for efficient lookups.
Formula (Cost): (M + (M * P_R * {probe cost}))
Example: (R(sid: 28)), probes (S)-index for (sid=28).
Implementation:
Scan (R) sequentially.
For each (R)-tuple, probe (S)-index (e.g., B+ tree) for matches.
Estimated Cost:
Read (R): **(M)** I/Os.
Probes: **(M * P_R * (Cost for finding Matching tuples))** I/Os (P_R: tuples/page, Cost for finding matching tuples: 1.2 for Hash Index, 2-4 for B+ tree index).
Index I/Os often cached.
Assumption: Index (e.g., B+ tree) exists on (S).
⚠️ Common mistake: Assuming efficiency without index.

Query Building

- *RA Expressions*: Combine operations into sequences.
Example: π_{name}(σ_{age > 30}(employee)) → names of employees > 30.
Expression Trees: Visualize queries.
Example:

```
π_name
   |
σ_age>30
   |
employee
```

Leaves: Relations; Root: Result.
Example Query: Bars on Maple St. or selling Bud < $3:
Formula: (π_{name}(σ_{addr = "Maple St."}(Bars)) ∪ π_{bar}(σ_{beer = "Bud" AND price < 3}(Sells)))

Set vs. Bag Semantics

- *Set Semantics*: No duplicates; RA default.
Example: π_{name}(employee) → one "John".
Drawback: Duplicate removal costly.

- *Bag Semantics*: Duplicates allowed; SQL default.
Union: Adds occurrences (e.g., {a,b,b} ∪ {b,c} = {a,b,b,b,c}).
Difference: Subtracts occurrences (e.g., {x,x,x,y,y,z} - {x,y,z,z} = {x,x,y}).
Advantage: Faster, supports streaming.
Example (Bag Difference):
Input: (A = {x,x,x,y,y,z}, B = {x,y,z,z})
Output: ({x,x,y}) using hash map.

Two-Way External Merge Sort
- *Two-Way Merge Sort*: Simplest external sorting method using 3 buffer pages (2 input, 1 output).

- **Process**:
Pass 0: Read 1 page, sort in memory, write as sorted *run*. Repeat for all pages.
Merge Passes: Merge 2 runs at a time into a larger sorted run until 1 run remains.

- **Formulas**:
Number of Passes: $\lceil \log_2 N \rceil + 1$. (N: pages)
Total I/O cost: 2N * ($\lceil \log_2 N \rceil + 1$)

- **Example**:
N=4: Pages [3,4], [6,2], [9,4], [8,7] → Runs [3,4], [2,6], [4,9], [7,8].
Pass 1: Merge to [2,3,4,6], [4,7,8,9].
Pass 2: Merge to [2,3,4,6,7,8,9].
Cost: 3 passes,
2×4×3=24

- **Key Point**: Simple but requires more passes, increasing I/O costs.

General External Merge Sort
General Merge Sort: Uses *B* buffer pages to create larger runs and merge more runs per pass.

Process:
- **Pass 0**: Read *B* pages, sort in memory, write as a *run*. Creates ⌈*N*/*B*⌉runs.
- **Merge Passes**: Merge *B*−1 runs at a time using *B*−1input buffers and 1 output buffer.
- **Formulas:**
> Number of passes: $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$
> Total I/O cost: $2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$

Typical External Merge Sort

QUERY OPTIMIZATION
What & Why?
• What: turn SQL/RA into an efficient physical plan
• Why: same logical query can vary hugely in I/O & CPU cost

Logical vs. Physical Plans
• Logical: tree of σ, π, ⋈, ∪, etc.
• Physical: choose an algorithm per operator (nested-loop, hash-join, sort-merge, etc.) + buffer/pipeline strategy

Pipelining vs. Materialize
• Pipeline: feed tuples directly to next operator—no temp writes
• Materialize: write intermediate to disk, then read back

"Good" Plans & Plan Space
• Join trees:
– Left-deep: outer input is a base table or prior join (easy pipelining)
– Right-deep / bushy: more parallelism, complex buffers
• 3 tables → 6 left-deep orders
• 5 join algs → 6×5^2 = 150 plans (before pruning)

Push-down Rules
1. Selection push-down: apply σ as early as possible
2. Projection push-down: drop unused cols at leaves
3. Push σ into indexed NL: e.g. A⋈σ(B.price>5)(B) uses B.id index

Cost Estimation (I/O)
• Scan/σ/π: B(R)
• Simple NL: B(R) + T(R)×B(S)
• Block NL: B(R) + ⌈B(R)/(B−2)⌉×B(S)
• Index NL: B(outer) + T(outer)×(index lookup)
• Sort-merge join (2-pass if B^2>pages):
- Sort R: 2·B(R)·[1+⌈log_{B−1}(B(R)/B)⌉]
- Sort S: same
- Merge: B(R)+B(S)
• Hash-join (M<$(B-2)^2$): partition 2M+2N, build/probe M+N ⇒3M+3N

2. TRANSACTION MANAGEMENT
What is a Transaction?
A sequence of SQL statements executed atomically (all or none).

Why Need?
Prevent inconsistency from crashes or concurrent updates
(e.g. \$20 transfer between accounts).

SQL Syntax
BEGIN TRANSACTION;

…SQL statements…
COMMIT;   -- or ROLLBACK

ACID Properties & Who Enforces
Atomicity - All-or-nothing execution - enforced by DBMS
Consistency - DB moves from one valid state to another - enforced by programmer (App logic + constraints)
Isolation - Concurrent Ts appear serial - enforced by DBMS
Durability - Once committed, changes survive crashes - enforced by DBMS

Locks & Concurrency Control
• Granularity: database > table > row > cell
• Pessimistic: lock before access
• Optimistic: validate at commit

Crash Recovery Intro
Primitive ops per T:
INPUT(X) : read page X into buffer
READ(X,t) : buffer X → local t
WRITE(X,t) : local t → buffer X
OUTPUT(X) : flush buffer X to disk

3. RECOVERY & CHECKPOINTING
Failures: system crashes (power loss) ⇒ memory lost, disk partial writes.
Assumptions: WAL ⇒ log records forced before data pages.
A) Undo-Logging
Log records:
<START T>
<T, X, oldValue>
<COMMIT T>
WAL rules:
1. Write <T,X,old> before OUTPUT(X)
2. Write <COMMIT T> after all OUTPUTs of T
Fuzzy checkpoint:
<START_CKPT,Active={T1…Tk}>
…flush pages of already-committed Ts…
<END_CKPT>
Undo recovery:
1. Backward scan to last <END_CKPT>, then to its <START_CKPT>; let C=Active.
2. Forward scan from <START_CKPT> to crash:
- Mark committed Ts seen
- Any T∈C w/o commit, and any new T′ w/o commit ⇒ ToUndo
3. Backward scan crash→<START_CKPT>:
- For each <T,X,old> with T∈ToUndo: X:=old
- On <START T>: remove T from ToUndo

B) Redo-Logging
Log records:
<START T>
<T, X, newValue>
<COMMIT T> -- may precede data writes
WAL rule:
Before OUTPUT(X), <T,X,new> & <COMMIT T> must be on disk
Fuzzy checkpoint: same markers as undo
Redo recovery:
1. Backward scan to last <END_CKPT>, then to its <START_CKPT>
2. Forward scan <START_CKPT>→crash: collect committed Ts ⇒ ToRedo
3. Forward scan again: for each <T,X,new> with T∈ToRedo: X:=new

C) Combined (ARIES-style) [optional]
1. Analysis: identify Dirty pages & active Ts
2. Redo: replay updates from earliest dirty LSN
3. Undo: rollback Ts still active at crash

1. Relational Algebra
(a) Two Derived Operations of Relational Algebra
Answer: Intersection and Join
(b) Expressing Intersection in Terms of Basic Operations
Answer: R - (R - S)

2. Implementation of Operators
(a) Union Operator Using Sort-Merge
Answer: Sort-merge union with cost 5M + 5N.
Explanation:
M, N < B(B-1), so sorting completes in two passes.
Sorting (External Sort):
- Pass 0: Divide R and S into sorted runs (read/write each page once → 2M + 2N).
- Pass 1: Merge runs into final sorted files (read/write again → 2M + 2N).
- Total sorting cost: 4M + 4N.
Merging:
- Merge sorted R and S, skipping duplicates (read each page once → M + N).
Total Cost: 4M + 4N (sorting) + M + N (merging) = 5M + 5N.

(b) Two Hash Functions in Hash Join
Answer: One hash function partitions relations into B-1 buckets; a different one builds in-memory hash tables for uniform distribution.
Explanation:
First Hash Function: Used to divide R and S into B-1 buckets on disk. Ensures tuples with matching keys (same join attribute) go to the same bucket.
Second Hash Function: Used within each bucket to build an in-memory hash table (tuned to RAM size). Ensures uniform distribution of tuples for fast lookups during probing.
Why Different: The first function optimizes for disk partitioning; the second for in-memory efficiency. Using the same function could lead to poor in-memory distribution, slowing lookups.

(c) Cost of Hash Join
Answer: Ideal cost is 3M + 3N.
Steps (Ideal Case):
Partitioning: Read R and S, hash into B-1 buckets, write back (read + write → 2M + 2N).
Build and Probe: Read each bucket of R and S into memory (fits due to M < (B-2)(B-2)), build hash table, and probe (read → M + N).
Total Cost: 2M + 2N (partition) + M + N (build/probe) = 3M + 3N.
Each page is read twice (partition, probe) and written once (partition), and buckets fit in memory.

(d) Cost of Sort-Merge Join for R and S
Answer: Total cost is 150 I/Os.
Explanation:
Relations:
- R (Enroll: sid, cid, grade) = 20 pages, S (Students: sid, sname) = 10 pages.
- Buffer = 6 pages, each page holds 100 tuples, each student in S enrolls in ≤3 courses.
Sort-Merge Join Steps:
> Sorting:
- R (20 pages) and S (10 pages) < B^2 (6^2 = 36), so sorting completes in 2 passes.
- Cost per relation: 2 passes × (read + write) = 4 × |pages|.
- R: 4 × 20 = 80 I/Os, S: 4 × 10 = 40 I/Os.
- Total sorting cost: 80 + 40 = 120 I/Os.
> Merging:
- Read sorted R and S, join on sid.
- Each student (tuple in S) matches ≤3 tuples in R (≤3 courses), fitting in one page.
- Cost: Read R (20 pages) + S (10 pages) = 30 I/Os.
Total Cost: 120 (sorting) + 30 (merging) = 150 I/Os.
Sorting is efficient (2 passes), and merging is straightforward since matches are small (≤3 per student).

3. **Query Optimization**
(a) 12 Physical Query Plans for SQL Query
Answer: 12 plans: 6 join orders (A→B→C, A→C→B, B→A→C, B→C→A, C→A→B, C→B→A) × 2 join algorithms (from simple nested-loop, block nested-loop, index nested-loop, hash join, sort-merge join). Push selections (B.price > 5, C.age > 30) and projections early.
Explanation:
Query: Joins A, B, C on A.id = B.id and B.id = C.id, filters B.price > 5, C.age > 30, projects A.name.
Plans:
Join Orders: 6 permutations (3! = 6).
Join Algorithms: Pick 2 of 5 (simple/block/index nested-loop, hash, sort-merge).
Total: 6 × 2 = 12 plans.

Optimizations:
Apply selections (B.price > 5, C.age > 30) before joins.
Project only required columns early.
Use join conditions (A.id = B.id, B.id = C.id, or A.id = C.id).
Purpose: Vary orders/algorithms to minimize I/O and CPU costs.

(b) Pushing Selection Below Indexed Join
Answer: Yes, push B.price > 5 below the join, as it doesn't affect the B.id index.
Explanation:
Context: A ⋈ B with indexed nested-loop join using B.id index.
Selection: B.price > 5 filters on price, not id.
Reason:
- Selection is independent of B.id, preserving index usability.
- Equivalent: σ_B.price>5(A ⋈ B) = A ⋈ σ_B.price>5(B).
Benefit: Reduces B's tuples before join, lowering cost.

4. **Transaction Management**
(a) What is a Transaction in Relational Databases?
Answer: A transaction is a sequence of SQL statements executed as a single atomic unit—either all succeed or none are applied.
Explanation:
Definition: A transaction groups SQL operations (e.g., INSERT, UPDATE) to act as one unit.
Atomicity: Ensures all statements complete successfully, or no changes are made (e.g., if one fails, all are undone).
Consistency: Maintains database validity—starting consistent, it ends consistent.
Purpose: Prevents partial execution, which could leave the database in an inconsistent state.
Execution: Transactions run sequentially in the application.

(b) Transaction V Interfering with Transaction T
Answer: Transaction V checks the account balance (SELECT checking + savings). Without ACID, V may read an inconsistent balance if it runs between T's subtraction from checking and addition to savings.
Explanation:
Transaction T: Moves $20 from checking to savings:
- Subtract $20 from checking.
- Add $20 to savings.
Transaction V: Queries total balance:
SQL: SELECT (checking + savings) AS total FROM Accounts WHERE acct_id = 123;
Interference:
- Without ACID (Isolation), V could run after T subtracts $20 from checking but before adding to savings.
- V would see a lower balance (missing the $20 not yet added to savings), causing inconsistency.
Why ACID Matters: Isolation ensures V sees either the state before or after T, not an intermediate state.

(c) How ACID Requirements Are Ensured
Answer:
Atomicity: DBMS uses transaction logs to undo partial changes if a transaction fails.
Consistency: Integrity constraints and rules enforce valid state transitions.
Isolation: Concurrency controls (e.g., locking) ensure transactions appear to run serially.
Durability: Write-ahead logging ensures committed changes are saved to stable storage.

5. **Recovery**

(a) Non-Quiescent Checkpointing
Answer: Non-quiescent checkpointing logs active transactions (<START CKPT (T1,…,Tk)>) without pausing the database, flushes committed dirty pages, and logs <END CKPT> when active transactions complete. The database does not freeze.
Explanation:
Concept: Records a checkpoint without stopping new transactions.
Process:
- Log <START CKPT (T1,…,Tk)> with active transactions.
Continue normal operations; flush committed dirty pages in the background.
Log <END CKPT> after T1,…,Tk commit or rollback.
Recovery: After a crash, start from last <END CKPT>, process only listed transactions and later ones.
No Freeze: Only log records are written atomically, allowing ongoing transactions.
Purpose: Reduces recovery time while maintaining consistency without halting the system.

(b) Undo Logging: Transactions to Undo After Crash
Answer: Undo active transactions (with <START T> but no <COMMIT T>) because their partial changes may be on disk, risking inconsistency.
Explanation:
- **Transactions to Undo**:
Active transactions at crash: Have <START T> but no <COMMIT T>.
Their changes may be on disk (undo logging allows early writes), leaving an inconsistent state.

- **Recovery Process**:
Scan log backward.
Restore old page values for these transactions' updates (rollback).

Why Not Others:
Committed transactions (<COMMIT T> logged) have all changes safely on disk.

Purpose: Ensures atomicity and consistency by removing partial, uncommitted changes.

6. **Recovery**
(a) Transactions Flushed After <start ckpt t5,t6>
Answer: T1, T4
Explanation:
Context: After logging <start ckpt t5,t6>, the system flushes changes by committed transactions.
Transactions:
T1: Committed (<commit t1>) before checkpoint.
T4: Committed (<commit t4>) before checkpoint.
T5, T6: Active during checkpoint (listed in <start ckpt t5,t6>), not yet committed.
T7: Starts after checkpoint, not relevant yet.

Why T1, T4:
Only committed transactions' changes (dirty pages) are flushed during checkpointing to ensure durability.
T5, T6 are active, so their changes are not flushed yet.
Purpose: Flushing T1, T4 ensures their committed changes are on disk for recovery.

(b) Recovery Actions Using Redo Log
Answer: Redo T7's update (<t7, z, 4>). Final values: x = 5, y = 3, z = 4.
Explanation:
Recovery Steps:
Locate Checkpoint: Find latest <end ckpt> (indicates completed checkpoint with T5, T6 active).
Identify Transactions to Redo:
- Transactions in <start ckpt t5,t6> that committed: None (T5, T6 have no <commit>).
- Transactions started after <start ckpt> and committed: T7 (<commit t7>).
Redo Updates:
- Apply T7's log entry: <t7, z, 4> → set z = 4.
- Flush changes to disk.
- **Ignore Uncommitted**: T5, T6 have no <commit>, so their changes (e.g., <t5, x, 2>) were not flushed to disk.
Final Values:
x = 5 (from T1's <t1, x, 5>, committed and flushed).
y = 3 (from T4's <t4, y, 3>, committed and flushed).
z = 4 (from T7's <t7, z, 4>, redone during recovery).
Why it works: Redo logging only applies committed changes; uncommitted ones are ignored as they weren't written to disk.