📚 Query Optimization Flow

Brief Overview

This note covers **SQL query optimization** and was created from a 22-page PDF presentation. It outlines the end-to-end flow from parsing to runtime, key logical rewrite rules, physical plan selection, and practical examples of join minimization.

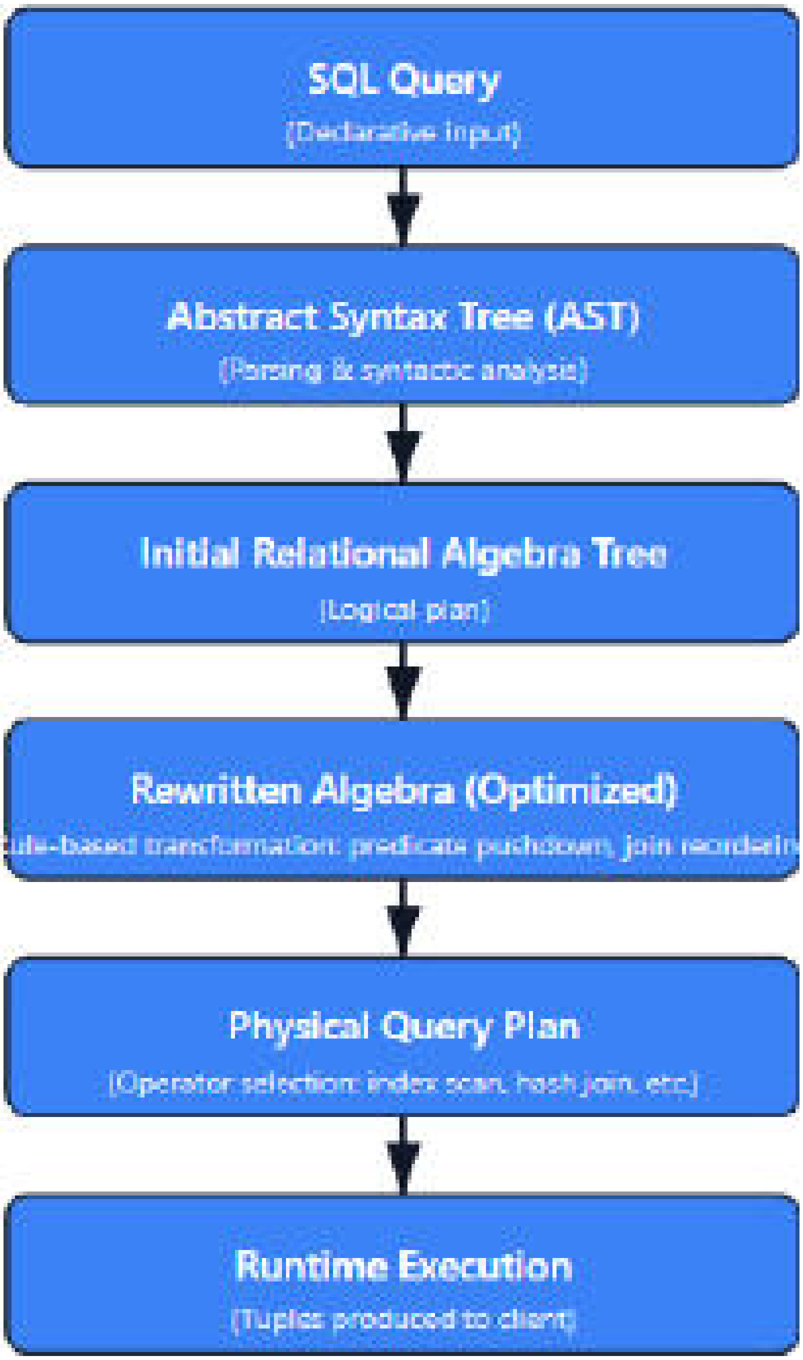
Key Points

- Understand the six-stage query lifecycle
- Master predicate push-down and join re-ordering
- Learn how logical rewrites influence physical plans
- See concrete examples with semi-joins and join minimization

---

📊 Overall Query Processing Flow

The flowchart below visualizes the six stages a SQL query goes through from textual input to result delivery.



The diagram shows a top-down progression:

1. **SQL Query** – declarative input.
2. **Abstract Syntax Tree (AST)** – parsing & syntactic analysis.
3. **Initial Relational Algebra Tree** – logical plan.
4. **Rewritten Algebra (Optimized)** – clause-based transformations (predicate push-down, join re-ordering).
5. **Physical Query Plan** – operator selection (index scan, hash-join, etc.).

6. **Runtime Execution** – tuples produced to the client.

🧩 Parsing & Semantic Analysis

- **SQL Query example**: SELECT name FROM Customer WHERE city = 'Paris';
- **Parsing → AST**

> The SQL string is tokenized and parsed into a tree representing grammatical components (SELECT, FROM, WHERE). Syntax errors are caught before semantic analysis.

- **Semantic Analysis → Initial Relational Algebra**

> The AST is translated into a logical algebra tree whose nodes are abstract operators (σ = selection, π = projection, $\bowtie$ = join). This captures the query's meaning independently of physical storage.

📐 Logical Algebra Generation

- **Logical operators** represent *what* to do, not *how*:
  - **σ** (selection) – filter rows.
  - **π** (projection) – keep required columns.
  - **$\bowtie$** (join) – combine relations.

The resulting tree is the **initial logical plan**.

🔄 Logical Rewriting (Optimization)

Principles

> **Equivalence Preservation** – every rewrite must return exactly the same result as the original query.
> **Relational Algebra as the Internal Language** – rewrites are applied at the algebra

> level.
> **Reduce Before You Join** – push selections (σ) and projections (π) as early as possible.
> **Local Transformations** – most rules modify small sub-trees.
> **Iterative Refinement** – apply rules repeatedly until a fixpoint is reached.

Heuristic Goals

- **Push selections down** → fewer rows enter joins.
- **Push projections down** → narrower tuples, lower memory use.
- **Reorder joins** using commutativity/associativity for cheaper execution orders.
- **Replace expensive operators** (e.g., nested subqueries → semi-joins).
- **Exploit constraints & keys** (PK/FK, uniqueness, nullability).
- **Remove redundant operators** (merge cascaded selections, drop unnecessary projections).

Common Rewrites

- Predicate push-down
- Join re-ordering
- Eliminate redundant projections/selections
- Convert IN / EXISTS subqueries to **semi-joins**

📚 Equivalence Rules

Rule Type	Example (Algebra)	Description
Commutativity	$\sigma_1 \sigma_2 R \equiv \sigma_2 \sigma_1 R$	Order of selections does not matter.
	$R \bowtie S \equiv S \bowtie R$	Join is symmetric.
	$R \cup S \equiv S \cup R$	Union is symmetric.
Associativity	$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$	Join grouping can be rearranged.
	$(R \cup S) \cup T \equiv R \cup (S \cup T)$	Union grouping can be rearranged.
Selection Rules	$\sigma_{c_1 \wedge c_2}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(R))$	Cascade of selections.

	$\sigma\_c(\pi\_{L}(R)) \equiv \pi\_{L}(\sigma\_c(R))$ if $\mathrm{attrs}(c) \subseteq L$	Move selection above projection when attributes match.
	$\sigma\_c(R \bowtie S) \equiv \sigma\_c(R) \bowtie S$ if c references only R	Predicate push-down through join.
Projection Rules	$\pi\_{L_1}(\pi\_{L_2}(R)) \equiv \pi\_{L_1}(R)$ if $L_1 \subseteq L_2$	Cascade of projections.
	$\pi\_{L_1 \cup L_2}(R \bowtie S) \equiv \pi\_{L_1}(R) \bowtie \pi\_{L_2}(S)$	Push projection below join (when safe).
Join-Selection Interaction	$R \bowtie\_{c}(S) \equiv \sigma\_c(R \times S)$	Theta-join equals selection over Cartesian product.
Set Operation Rules	$\sigma\_c(R \cup S) \equiv \sigma\_c(R) \cup \sigma\_c(S)$	Selection distributes over union.
	$\pi\_L(R \cup S) \equiv \pi\_L(R) \cup \pi\_L(S)$	Projection distributes over union.
Rename (ρ) Rules	$\rho\_{A \to B}(\sigma\_c(R)) \equiv \sigma\_{c'}(\rho\_{A \to B}(R))$	Rename commutes with selection (attributes in c renamed).
	$\rho\_{A \to B}(\pi\_L(R)) \equiv \pi\_{L'}(\rho\_{A \to B}(R))$	Rename commutes with projection.

🛠️ Physical Plan Selection

Logical operators are mapped to concrete algorithms:

Logical Operator	Physical Alternatives
Scan	*Seq Scan* (full table) vs. *Index Scan* (use B-tree/bitmap index)
Join	*Nested Loop Join* (row-by-row), *Hash Join* (build hash table), *Merge Join* (sorted inputs)
Aggregation	*Hash-aggregate*, *Sort-based aggregate*

Sorting	*External merge sort, In-memory quicksort*

The chosen algorithms form the **physical query plan**, a blueprint for execution.

🚀 Execution Engine (Runtime)

- Operators run as **iterators**, pulling tuples from child nodes.
- Intermediate results stream **bottom-up** (leaf → root).
- Final rows are returned to the client application.

🔗 Logical ↔ Physical Optimization Connection

- **Logical rewriting shrinks the search space** for physical plans by removing unnecessary operators and reducing intermediate cardinalities.
- Early **selection/projection push-down** → smaller intermediate results → fewer physical join candidates → lower I/O & memory use.
- A **poor logical plan** (e.g., bad join order) forces the physical optimizer to consider expensive join strategies; a **good logical plan** enables cost-based choices (hash, merge, index-nested-loop) on a simpler tree.

📈 Semi-Join Concept

> A **semi-join** R ⋉ S returns all tuples of R that have at least one matching tuple in S; it does **not** append columns from S.

Key properties

- Produces only the left-hand relation's attributes.
- Implements existence checks (IN, EXISTS).
- Reduces tuple width and intermediate result size.
- Enables join reordering and selection push-down across subquery boundaries.

🔧 Join Minimization

Goal: Eliminate joins that do not affect the result.

- **Redundant join**: when the information it supplies is already guaranteed by other joins or constraints (PK/FK, functional dependencies).
- **Benefits:** fewer intermediate results → faster execution.

Typical sources of redundancy

- Hand-written queries or auto-generated SQL.
- Unfolded view definitions.
- Schemas with strong constraints.

Pattern folding (homomorphism) – mapping multiple variables to a single one without altering semantics.

📝 End-to-End Query Processing Examples

Example 1 – Customer / Order

SQL

```
SELECT cname
FROM Customer c
WHERE EXISTS (
    SELECT 1
    FROM OrderInfo o
    WHERE o.cid = c.cid
      AND o.total_value > (
          SELECT AVG(o2.total_value)
          FROM Customer c2
          JOIN OrderInfo o2 ON c2.cid = o2.cid
          WHERE c2.city = c.city
      )
);
```

Logical Rewrite Highlights

- **Selection push-down** on OrderInfo (filter by cid).
- **Projection push-down** (keep only needed columns).
- **Join re-ordering** to compute city-wise average before applying the outer filter.
- **Subquery to semi-join** (EXISTS → semi-join).
- **Removal of unused attributes** (e.g., columns not needed for final cname).

Resulting logical plan (simplified):

```
π_cname ( σ_{ o.total_value > AvgCity } ( Customer ⋈ OrderInfo ) )
```

where AvgCity is computed via an aggregation on Customer ⋈ OrderInfo grouped by city.

Example 2 – Student / Course / Dept

SQL

```
SELECT s.sname
FROM Student s
JOIN Enroll e ON s.sid = e.sid
JOIN Course c ON e.cid = c.cid
WHERE c.dept IN (
    SELECT d.dept FROM Dept d WHERE d.chair LIKE '^Dr\.'
)
AND c.credits > (
    SELECT AVG(c2.credits) FROM Course c2 WHERE c2.dept = c.dept
);
```

Logical Rewrite Highlights

- **Selection push-down** on Dept (chair LIKE '^Dr\.').
- **Projection push-down** (retain only dept).
- **IN → semi-join**: c.dept ⋉ Dept_filtered.
- **Join minimization** – join Course with the filtered Dept first, then with Enroll, and finally with Student.
- **Aggregation push-down** – compute AVG(credits) per department before applying the credit filter.

Resulting join order (optimal):

```
((Course ⋉ Dept_filtered) ⋈ Enroll) ⋈ Student
```

📊 Summary Tables

Logical vs. Physical Optimizations

Aspect	Logical Optimization	Physical Optimization
Focus	Transform algebraic expression while preserving semantics.	Choose concrete algorithms for each operator.
Typical Techniques	Predicate push-down, projection push-down, join re-ordering, semi-join conversion.	Index scan vs. sequential scan, hash vs. merge join, parallel execution.
Effect on Search Space	Reduces size of algebra tree → fewer physical alternatives.	Explores cost-based choices on the reduced tree.
Primary Goal	Lower cardinalities & tuple widths.	Minimize I/O, CPU, and memory usage.