ChatGPT

Taxonomy of Semantic SQL Optimization Patterns and LLM Validation Harnesses

Part 1: Semantic Anti-Pattern Library

This section catalogs **15 common SQL anti-patterns** where query logic is redundant or inefficient despite being syntactically correct. Each anti-pattern includes a description of the human error, how to detect it (trigger), the state-of-the-art (SOTA) rewrite using modern SQL, and metadata "safety checks" an AI agent should perform to ensure the rewrite preserves query semantics.

1. **The Double-Dip (Repeating Subqueries)**
2. **Human Error:** Writing the same subquery or calculation multiple times (e.g. once in a `SELECT` and again in a `WHERE`), causing redundant computation. This "don't repeat yourself" violation often happens when a developer copy-pastes a subquery in multiple places.
3. **Semantic Trigger:** Identical subquery or `SELECT` block appearing more than once in the query (e.g. the same `SELECT AVG(...) FROM ...` used in multiple clauses). A telltale sign is a complex subquery whose text is repeated, or identical filter logic in multiple sub-selects.
4. **SOTA Rewrite:** Use a **Common Table Expression (CTE)** or derived table to define the subquery once and reference it multiple times [1]. Alternatively, compute the value in one place and join or filter by it as needed. For example, instead of writing a subquery twice, use `WITH subQ AS (...) SELECT ... FROM subQ JOIN ... WHERE ...` so that the subquery executes once [1]. This eliminates duplicate work and improves readability.
5. **Safety Check:** Ensure that the CTE or derived table is **semantically identical** to the repeated subquery and does not alter execution order in a way that changes results. Verify that the CTE doesn't introduce unintended duplicate rows (check primary key or unique constraints on the CTE results if applicable). Confirm the DBMS will not *re-materialize* the CTE per usage (in most modern engines, identical CTE usage is optimized, but older versions might not) – if needed, test on a small dataset to ensure one execution. Also, check that moving the subquery's logic into a CTE doesn't change scoping (especially if the subquery was correlated and relied on an outer query reference – in that case, a direct CTE might turn a correlated subquery into a cross join; make sure to include the correlation columns in a join condition when refactoring). Essentially, the agent should use schema knowledge (keys of the subquery result) to ensure one-to-one merge with the outer query.
6. **The Row-by-Row Subquery (Correlated Scalar Query instead of JOIN)**
7. **Human Error:** Using a correlated subquery in the `SELECT` or `WHERE` clause to fetch a value for each row of an outer query. This yields the correct result but can be extremely slow because the subquery executes for every row of the outer table [2] [3]. For example, selecting a customer's total orders by doing `(SELECT COUNT(*) FROM orders WHERE orders.cust_id = customer.id)` for each customer.
8. **Semantic Trigger:** A **correlated subquery** referencing the outer query's columns, often introduced with `WHERE outer.id = inner.id`. If the subquery returns a single value per outer row, it's a scalar subquery; if used with `EXISTS` or `IN`, it's checking set membership.

The pattern is indicated by keywords like `SELECT ... FROM ... WHERE ... = outer.table.col` inside an outer query.

9. **SOTA Rewrite:** Replace the correlated subquery with a **JOIN or LEFT JOIN** (or a derived table/CTE) so that the database can retrieve all needed data in one set-oriented operation [4]. For instance, join the `orders` table and aggregate counts per customer instead of counting via subquery for each customer. Modern optimizers often convert simple cases automatically, but explicit rewrites using joins or **window functions** are clearer. In the above example, a rewrite would be: `SELECT c.id, COUNT(o.*) FROM customers c LEFT JOIN orders o ON o.cust_id = c.id GROUP BY c.id`. This computes all counts in one pass [4]. If the original subquery was scalar (e.g., selecting a single related value), use an appropriate join and aggregation (or `MAX/MIN` for one-to-one relations) or ensure the join doesn't duplicate rows (see Safety Check).

10. **Safety Check:** Confirm that the join is **semantically equivalent** to the subquery. If the subquery inherently returns at most one row per outer row (e.g., it's selecting from a one-to-one related table or using an aggregate without GROUP BY on a foreign key reference), ensure there is a unique index or primary key on the join key in the inner table [5]. This guarantees the join won't multiply rows. If the subquery used an aggregate (like SUM or COUNT), ensure the join + aggregation yields the same result – check for any additional filtering in the subquery that must be applied to the joined dataset. Use foreign key metadata and uniqueness constraints: for example, if rewriting `SELECT (subquery returning one value) FROM A` to a join `A JOIN B`, verify that B's joining column is a primary key or the subquery applied an aggregate so that duplicates in B don't break equivalence. If the correlated subquery had an `EXISTS` condition, the join rewrite should use a DISTINCT or an appropriate filtering to avoid duplicates. The agent should only rewrite when it can confirm one-to-one or aggregate equivalence; otherwise, a naive join could change result counts.

11. **The Manual Pivot (Self-Join for Conditional Aggregates)**

12. **Human Error:** Querying the same table multiple times with different filters or conditions to simulate a pivot or conditional aggregation. For example, to count orders by status, a user writes `SELECT (SELECT COUNT(*) FROM orders WHERE status='NEW'), (SELECT COUNT(*) FROM orders WHERE status='SHIPPED'), ...` or by self-joining `orders` table to itself filtering each copy by status. This approach repeats scanning the table for each condition, which is redundant.
13. **Semantic Trigger:** Multiple subqueries or joins against the **same table** with varying filter conditions. Typically, you'll see the same base table alias repeated (or multiple aliases of the same table) in the `FROM` clause, each with a condition in the `WHERE` that should be mutually exclusive or related to a specific category. Or you see multiple scalar subqueries in `SELECT` that differ only by a literal in a condition.
14. **SOTA Rewrite:** Use **conditional aggregation or FILTER/LATERAL** features instead of repeated scans. Modern SQL supports `SUM(CASE WHEN status='NEW' THEN 1 ELSE 0 END)` or the ANSI SQL `FILTER` clause in aggregations, or even **pivoting** if available. For example: `SELECT COUNT(*) FILTER(WHERE status='NEW') AS new_count, COUNT(*) FILTER(WHERE status='SHIPPED') AS shipped_count FROM orders;`. Alternatively, a single pass with **GROUP BY** and then a pivot in application logic (or using `MAX(CASE ...)` technique) can replace multiple joins. If multiple metrics are needed (counts, sums, etc.), use one grouped subquery or CTE per table instead of one per metric.

15. **Safety Check:** Validate that each self-joined table or subquery was independent (no interdependency that would be lost when combining). If the self-joins were meant to enforce

some relational condition, combining might alter logic. Ensure that the combined single-query approach doesn't **double-count** or miss data: e.g., in the pivot example, the conditions should be disjoint. Check that no rows satisfy multiple conditions unless the original logic counted them multiple times (if so, the combined approach should replicate that intentionally or the original was likely wrong). Use domain knowledge (like status categories) or check constraints to verify disjointness. Also confirm the rewrite does not introduce `NULL` handling issues – for instance, a self-join might implicitly exclude nulls in a condition, whereas a CASE expression would include them unless handled. So, apply the same filters in the CASE or FILTER clauses. Essentially, the agent should use schema info (check constraints, enumerations, etc.) to ensure the unified query matches the logic of repeated parts.

16. **Greatest-N Per-Group via Self-Join (The "Top-N Self-Join" Anti-pattern)**

17. **Human Error:** Finding a maximum (or minimum) per category by joining a table to itself with inequality conditions. For example, to find each department's highest salary, a user might self-join the employee table as `e1 JOIN e2 ON e1.department = e2.department AND e1.salary < e2.salary` and then look for rows where `e2.salary IS NULL` (meaning e1 had no larger salary in its dept). This classic greatest-per-group solution works but is slow and complex [6] [7]. It effectively compares each row to many others.
18. **Semantic Trigger:** A **self-join with non-equi conditions** (e.g. `<` or `>`) and a pattern of checking for `NULL` on the second table's side, or using aggregation on the second table. Often combined with `GROUP BY e1.id` or a `HAVING COUNT(*)` logic (like another approach counts how many others are bigger, etc.). If you see `e1.X < e2.X` (or `>`), that's a hint.
19. **SOTA Rewrite:** Use **window functions** like `ROW_NUMBER()` or `RANK()` partitioned by the grouping key [8]. For example: `SELECT * FROM (SELECT department, name, salary, ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) as rk FROM employees) sub WHERE rk = 1;` to get the top salary per department. Window functions are optimized for this use-case and far more readable [8]. Another modern option is using a **LATERAL join** (or cross apply) to a subquery that selects the top N per group (if supported by the SQL dialect, e.g., PostgreSQL's `DISTINCT ON` or TOP clause with a join).

20. **Safety Check:** Confirm that the window function approach yields the *same rows* as the self-join method. If the original query did ties handling (some self-join patterns pick all max-tie rows, others pick one arbitrarily), ensure the window specification (RANK vs ROW_NUMBER) and filtering replicate that. Check for any additional filters in the original that must be applied either before or after the window calc. Also, ensure the ordering in the window function exactly matches the self-join's logic (e.g., highest salary, perhaps secondary tiebreakers if any). From metadata, ensure that the partition key (department in example) matches exactly – if department was a foreign key to a department table, the presence of all departments in the result might differ if some department had no employees (the self-join approach inherently returns nothing for empty departments, the window approach similarly would return none, which is fine). No special metadata like constraints is typically needed for correctness here, but the agent should verify that the self-join's exclusion of certain data (like `NULL` values perhaps) is mirrored. Additionally, check performance metadata: ensure the table is not extremely small (in which case performance differences are minor) or extremely large (where the self-join would be a huge hit – though that just reinforces doing the rewrite).

21. **Unnecessary DISTINCT (Distinct to Remove Join Duplicates)**

22. **Human Error:** Using `SELECT DISTINCT` to eliminate duplicate rows that result from a poor join or query design. Often, developers add DISTINCT as a quick fix when they see duplicate

output, without realizing it may mask an underlying join condition issue or one-to-many relationship. This can be inefficient because it forces a sort/aggregation to remove duplicates, and it may hide logical errors.

23. **Semantic Trigger:** Presence of `SELECT DISTINCT` in a query that joins multiple tables, especially if one of the tables is on a one-to-many relationship. For example, joining customers to orders and then using DISTINCT on customer fields, or joining and then grouping solely to eliminate duplicates. Another clue is a query where every column is functionally determined by a subset (like including a primary key from one table and still using DISTINCT – that implies redundant join).
24. **SOTA Rewrite:** Identify the **root cause of duplicates** and rewrite the query to avoid producing them in the first place. If the join is duplicating data (e.g., joining a one-to-many and you only needed one side), consider using an `EXISTS` or semi-join instead of a normal join [9] [10]. For instance, to get all customers who placed orders, one might do a join and DISTINCT – instead, use `SELECT customer.id FROM customers c WHERE EXISTS(SELECT 1 FROM orders o WHERE o.cust_id = c.id)`. This returns each customer at most once by design. Another approach is **aggregation** if you intended some aggregation – e.g., if duplicates came from multiple detail rows, use `GROUP BY` on the primary key with aggregate functions for detail info as needed. The key is to **refactor the join**: either add the missing join condition (if it was an error) or change the query structure so that multiplicative effects are handled (exists, max, etc., depending on intent). In modern SQL, using window functions or QUALIFY (in databases that support it) can also filter duplicates without a full DISTINCT.

25. **Safety Check:** Ensure that removing DISTINCT does not change the **result set semantics**. This requires confirming that the new approach truly yields one record per entity as intended. Use primary/foreign key metadata: e.g., if a DISTINCT was used because of a one-to-many join, verify the one-to-many relationship via foreign key (customer->orders) and ensure the new EXISTS approach returns exactly the customer list without duplication – by design it will. If the original query actually needed the detail data but just wanted to collapse duplicates, ensure the rewrite still returns the correct aggregated or representative detail. Check functional dependencies using schema (if table B has multiple matching rows per A, decide which one should be represented or if any differences matter). If the original DISTINCT was hiding a bug (like missing join condition causing cartesian blow-up), the rewrite might produce fewer rows – which is *correct* logically but might differ from the original buggy output. The agent should confirm the user's intent via constraints. For instance, if two tables should have been related by both col1 and col2 (composite key) but the join used only col1, it created duplicates; the DISTINCT removed them arbitrarily. A safe rewrite is to fix the join condition to include col2 (using metadata about keys) so that duplicates don't arise [11]. This both avoids needing DISTINCT and preserves true intent. Always verify that any condition used to reduce duplicates (like an EXISTS or an extra join predicate) doesn't exclude legitimate data.

26. **OR-Chain Query (Multiple ORs Instead of IN or UNION)**

27. **Human Error:** Filtering on multiple values by writing a long `OR` chain, e.g. `WHERE col = 'A' OR col = 'B' OR col = 'C'...`. In queries with many values, this is verbose and can prevent efficient index usage (some optimizers might not handle many ORs well). Similarly, writing separate `SELECT` queries combined by `UNION` for each value is an even worse variant [12] [13].
28. **Semantic Trigger:** A `WHERE` clause with a series of `OR` conditions on the same column (or a small set of columns), or multiple `SELECT ... WHERE ...` blocks unioned together each with

one condition. If you see patterns like `col = X OR col = Y OR col = Z`, that's a trigger. Or a query text repeating the same base `SELECT` with different literals, joined by `UNION` [14] [13].

29. **SOTA Rewrite:** Use the `IN` operator for clarity and to allow the optimizer to potentially use a hashed lookup or index range. For example, `WHERE col IN ('A','B','C')` expresses the same filter [15]. The optimizer can often turn that into a range scan or set membership check which is efficient. If the logic is more complex (like ranges or combinations), ensure parentheses properly group conditions, or consider a temporary table of values to join against for very large lists. In some cases, replacing ORs with a **JOIN on a values list** (e.g., using a CTE or table-valued parameter containing the values) can be optimal. If the query was written as separate `UNION ALL` queries per value, definitely refactor to a single query with an IN (or a UNION of fewer subqueries if they were hitting different tables). Note: For range conditions combined with OR (like `col < 3 OR col > 5` in the example), rewrite to two range conditions still combined by OR is fine, but ensure no overlap. If each OR block was on a different column, consider logically separating those (that's a different scenario involving OR across columns which might need parentheses – see pattern 7 below).

30. **Safety Check:** This rewrite is usually straightforward equivalence-wise, but the agent should be cautious about **NULL logic** and data types. `col IN (...)` is semantically the same as a series of ORs for equality, so result sets won't change. However, ensure that all OR clauses indeed were equality checks on one column (or an equivalent set of columns). If the ORs mix different columns or conditions (like `col = 5 OR status = 'X'`), then IN is not applicable directly and another approach is needed (or parentheses to clarify logic). Also, verify that combining into one query doesn't change how duplicates are handled: for example, if the original used `UNION` (which deduplicates results) vs `UNION ALL`, switching to a single query with IN will naturally return duplicates if an underlying table has them. If the original intended to deduplicate, maybe it used `UNION` instead of `UNION ALL` – in that case, ensure to add DISTINCT if needed (though typically filtering by IN on one table doesn't by itself introduce duplicates beyond what data has). Use the query plan or indexes: check that an index exists on the filtered column – if so, the IN list will use it efficiently; if not, the OR and IN are equally doing full scans. From a correctness perspective, no metadata is usually required beyond confirming the column names and types are the same for all OR'd conditions. If the values list is extremely large, consider that an IN list might hit a parameter limit or be less readable – in such cases, a temporary table or such could be considered, but that's an optimization nuance beyond semantics.

31. **Unnecessary UNION (Separate Queries for Conditions)**

32. **Human Error:** Writing separate queries combined with `UNION` when a single query with a broader `WHERE` condition would do. For example, querying the same table twice: `SELECT * FROM sales WHERE region='East' UNION SELECT * FROM sales WHERE region='West';`. If `UNION ALL` is used, it's clearly just appending results that could be achieved with an OR. If `UNION` (distinct) is used, it might indicate concern about duplicates, but often those duplicates arise only because the queries aren't mutually exclusive or are hitting the same data. This pattern is inefficient because it runs multiple scans [14].
33. **Semantic Trigger:** Multiple `SELECT` statements selecting from the **same table** or equivalent sets, differing only in filter conditions, connected by `UNION` or `UNION ALL`. Or even if different tables, if those tables are similar (like partitioned tables that could be addressed with one query via a partition key). Look for the repeated FROM clause or very similar SELECT clauses.
34. **SOTA Rewrite:** If they truly query the same table, consolidate them with a single `SELECT` and combine conditions with `OR` or `IN` as appropriate (similar to pattern 6). In the above example, `SELECT * FROM sales WHERE region IN ('East','West')` covers both [15]. If `UNION` (distinct) was used unnecessarily (because results were inherently distinct or because the author

wasn't sure), default to `UNION ALL` or a single query with proper conditions. Only use `UNION` when you need to eliminate duplicates across different result sets. In modern SQL, if these were meant to address different partitions or tables, consider using **partition union** methods (like a partitioned view or set operation within the engine) – but since we assume the same table, that's likely not needed.

35. **Safety Check:** Verify that the separate queries truly produced disjoint result sets or that duplicates are acceptable. If the conditions in each subquery are mutually exclusive (e.g., region East vs West), then `UNION ALL` is equivalent to OR and no duplicate elimination is needed. The agent should check for overlapping conditions – if there is any overlap, `UNION ALL` would double-count those overlaps, whereas `UNION` would not. In that case, the single-query approach with OR will naturally handle overlaps correctly (because each row either meets the combined condition once). If the original used `UNION` to be safe, confirm via constraints that overlaps cannot occur (e.g., a row cannot simultaneously satisfy both conditions). If overlap is possible and was not intended to be counted twice, ensure to keep distinct semantics. Typically, though, if it's the same table and same filters just different values, overlaps mean the conditions were not actually exclusive (like `region='East' OR region='West'` covers all East and West – no overlap since a row can't be both East and West). So it's safe. Check that the SELECT clauses are identical – if not, merging queries might require adding missing columns with NULLs for alignment, which complicates semantics. But in our scenario of identical FROM and only differing filters, it's straightforward. No special metadata is needed beyond confirming the query structures. Performance metadata (like table row count) can inform that doing one combined scan is cheaper than two scans, but that's obvious logically.

36. **SELECT*** (Retrieve All Columns Unnecessarily)

37. **Human Error:** Using `SELECT *` in production or complex queries where not all columns are needed. This is not a logical error per se – the results are correct – but it's a semantic anti-pattern in that it pulls more data than necessary, can break if schema changes, and can disable certain optimizations (like covering index usage). It is often just a lazy shortcut that can harm performance as the dataset or network load grows.
38. **Semantic Trigger:** The literal `*` in the select list of a query that joins multiple tables or in subqueries. Especially if followed by other explicit columns or with complex joins, it likely means the user didn't bother to specify needed fields. Another clue: use of `SELECT *` in subqueries that are later joined – the subquery might be pulling all columns when only one or two are used in the join or outer query.
39. **SOTA Rewrite:** Explicitly project only the needed columns. For a given query, determine which columns are actually used downstream (in application or later in the query) and list them out. This not only reduces I/O but also makes the query **self-documenting**. For example, replace `SELECT * FROM orders JOIN customers...` with `SELECT orders.id, orders.date, customers.name, ...` as needed. In cases of subqueries, select only the join key or aggregate needed instead of `*`. Modern IDEs and tools can sometimes fill the column list for you to edit. Another advanced approach: if an LLM agent is automating this, it could use schema info to expand `*` and then remove columns not necessary for correctness (though determining "necessary" might require knowledge of how results are consumed). Regardless, the best practice is **do not use `*` in permanent or performance-sensitive queries** [16].

40. **Safety Check:** Listing specific columns doesn't change which rows are returned, but it does change the shape of the result. The agent must verify that the consuming context (e.g., a calling application or a parent query) does not rely on `*` bringing all columns. If this is purely an optimization step by the agent, it should know which columns are actually needed. In an

autonomous setting, the agent might use **data flow or explain plans** to see which columns are used by later operations (e.g., if the next step only does `COUNT(*)`, then we actually don't need any columns at all – we could do `COUNT(1)`). Ensure that removing columns doesn't drop something required for correctness (for instance, some queries use `SELECT *` in a CTE assuming later they might need any column; the agent should be careful if it cannot infer usage). Also, confirm no side-effects: in some DBMS, selecting less data might change locking (though typically not). Mainly, the agent should check that no calculated or derived column is lost. Since this is within a single query, information_schema won't directly say which columns are needed; instead, a conservative approach is to keep all columns referenced elsewhere (JOIN/WHERE/GROUP BY/HAVING/ORDER BY clauses) plus those known to be required output. If uncertain, it might leave `*` to avoid dropping needed data. But if we assume an AI integrated into development, it knows what the user is ultimately asking. In summary, ensure the projection covers all necessary data. No integrity constraints are involved here, but **column alias** usage should be preserved if the query expects certain names.

41. **Premature Ordering (ORDER BY in Subqueries without LIMIT)**

42. **Human Error:** Sorting data in a subquery or view where the order is not used by the outer query, or will be resorted again. For example, `SELECT * FROM (SELECT ... FROM table ORDER BY col) sub;` without a `LIMIT` clause – the inner ORDER BY is wasted work because SQL results are unordered unless explicitly ordered at the final output [17] [18]. This often comes from misunderstanding that a sorted subquery will deliver sorted rows to the outer query (which is not guaranteed unless the outer does an ORDER BY too).
43. **Semantic Trigger:** An `ORDER BY` clause inside a subquery, view, or CTE that is not paired with a limiting clause (`TOP, LIMIT, FETCH FIRST N`) or needed for a subsequent operation (like a window function). If the outer query also orders, then the inner one is redundant or conflicting. In general, any inline view/CTE with `ORDER BY` and no consumption of that ordering is a red flag.
44. **SOTA Rewrite:** Remove the unnecessary `ORDER BY` from subqueries or push it to the top-level query only. SQL guarantees result ordering only with a top-level ORDER BY, so intermediate sorts should be done only if needed for correct logic (e.g., when using `ROW_NUMBER()` or aggregate functions that depend on order within a window). In modern SQL, if you need the top-N pattern, you keep the ORDER BY with a LIMIT in the subquery. Otherwise, drop it. The rewrite is straightforward: omit the `ORDER BY` in subselects that don't feed into a LIMIT or a function. This relies on letting the optimizer choose the best join/aggregation order, which it will.

45. **Safety Check:** Ensure that removing the ORDER BY does not change the semantics of the query **results**. Normally it shouldn't, because ordering isn't supposed to affect set equality of results – but if the subquery was using `ORDER BY` to, say, make a nondeterministic function stable or to get consistent results with `LIMIT`, then dropping it could change which rows are returned. For example, if someone did `SELECT * FROM (SELECT * FROM table ORDER BY random() LIMIT 10) t;`, removing the ORDER BY random() would break the logic. So the agent should check if the `ORDER BY` is paired with a `LIMIT` (or a window function like `ROW_NUMBER() OVER (ORDER BY ...)` in the subquery). If so, it might be necessary for correctness (with LIMIT) or partially needed (with window, the ordering is needed but should be kept within the window function, not as a separate clause). If there is a `LIMIT` in the subquery, **do not remove the ORDER BY** – instead, consider moving the whole top-N logic up or leaving it as is. If there's no LIMIT and no obvious reason (like a comment or pattern of use in a window function) for the sort, it's safe to remove. From a metadata perspective, none is required, but the agent should maybe inspect the outer query to see if it applies its own ORDER BY (making inner one

redundant) or if the inner ORDER BY was a misguided attempt at relational ordering. After removal, if the final result needs ordering, ensure an ORDER BY at the end is present (the agent could add one if the user originally might have expected sorted output due to the subquery's presence). Essentially, maintain the contract that final output ordering should only come from final ORDER BY.

46. **Non-Sargable Predicates (Functions or Ops on Indexed Columns)**

    - **Human Error:** Applying transformations to columns in filters or join conditions such that indexes cannot be used. For example, `WHERE DATE(order_date) = '2024-01-01'` or `WHERE salary*1.1 > 50000` – wrapping the column in a function or arithmetic forces the database to evaluate it for every row, often bypassing indexes [19] [17]. This is logically correct but can be optimized. Similarly, joining on expressions (e.g., `JOIN ... ON SUBSTR(a.code,1,3) = b.prefix`) has the same issue [17].
    - **Semantic Trigger:** Any `WHERE` clause or `JOIN ON` that contains a function call or arithmetic on a column name (left side of comparison). Common ones: `UPPER(column) = 'X'`, `column + 0 = value`, `CAST(column AS ...) = ...`, or using non-leading wildcards in LIKE (`WHERE col LIKE '%abc'` is non-sargable because of the leading `%`). If an index exists on that column, these patterns prevent index seeks. The presence of these patterns strongly suggests an anti-pattern unless the dataset is small or the function usage is unavoidable.
    - **SOTA Rewrite:** Rewrite the predicate to avoid manipulating the column. Instead, transform the **literal/parameter** if possible, or use a range condition. For the date example, if `order_date` is a datetime, write `WHERE order_date >= '2024-01-01' AND order_date < '2024-01-02'` to cover that date range [19]. This way, an index on `order_date` can be utilized. For string cases like `UPPER(name) = 'ABC'`, consider storing an uppercase version of the column or using a case-insensitive collation/index. Alternatively, use a generated/indexed expression if supported. For joins on expressions, see if the logic can be refactored: e.g., if `SUBSTR(a.code,1,3) = b.prefix`, perhaps ensure both tables store the prefix (maybe add a computed column on `a` or join on `a.code LIKE b.prefix||'%'` with caution). The goal is to make comparisons on raw columns. Another strategy: push the function to the other side of comparison if that side is a single value. E.g., instead of `FUNCTION(col) = value`, do `col = INV_FUNCTION(value)` if `INV_FUNCTION` is the inverse (not always possible, but for simple math like `col+5 = 10` rewrite to `col = 5`). For patterns like `%abc` in LIKE, the only improvement might be full-text indexes or avoiding the leading wildcard (if requirements allow).
    - **Safety Check:** Verify that the rewritten predicate is **logically equivalent** to the original. For range transformations of functions, ensure the boundaries are correct (inclusive/exclusive as needed). The date example must cover the exact same times as the original function. Off-by-one errors would change results. If rewriting `UPPER(col) = 'ABC'` to `col = 'abc'` with case-insensitive collation, ensure the database or session is in a case-insensitive mode or that an index with that collation exists (metadata: collations, index definitions). For arithmetic, be mindful of rounding or data type differences (e.g., if `col` is integer, `col * 1.1 > 50000` can be rewritten to `col > 50000/1.1`, but if `col` is int, division might need ceiling). Use data type info from information_schema (numeric vs int vs float) to handle such edges. In join conditions, adding a computed column or an indexed expression changes schema – in an automated rewrite, the agent likely can't add schema elements on the fly, so it may instead recommend such an index to the user. If rewriting the query only, maybe it can use a subquery to precompute

needed values on the smaller table (which is a form of lateral join usage). Any such refactor should be checked against constraints: e.g., does the rewrite restrict nulls properly? (If original was `WHERE FUNC(col) = X`, if `col` is NULL, original predicate is FALSE/UNKNOWN; the rewritten might implicitly treat NULL differently if not careful). Typically, the agent should ensure to preserve the null logic (maybe explicitly add `AND col IS NOT NULL` if needed to mimic the original function returning null). Also, confirm that moving a predicate from one side to the other doesn't alter results when one side is null or if precision issues occur. This pattern is mostly about performance, but correctness must remain identical.

47. **Misplaced Outer Join Filter (Turning Outer Join into Inner Join)**

    - **Human Error:** Applying a `WHERE` filter on a column from the outer-joined table (the optional side), effectively nullifying the outer join. For example, `SELECT * FROM A LEFT JOIN B ON A.id=B.id WHERE B.value = 5;`. The intention might have been to get all A and mark those with B.value=5, but by putting `B.value=5` in WHERE (instead of the ON or using an IS NULL check), the outer join behaves like an inner join [20] [21]. This is a logical bug *and* inefficiency, as it confuses readers and possibly the optimizer.
    - **Semantic Trigger:** A query with `LEFT JOIN` (or RIGHT JOIN) where a condition on the right-side table appears in the `WHERE` clause without an `IS NULL` check to preserve non-matching rows. If the `WHERE` includes something like `B.column = someValue` or `B.column IS NOT NULL`, it's filtering out all the NULL-extended rows from the outer join, thus equivalent to an inner join. Another sign is a pattern `... LEFT JOIN ... WHERE ... OR B.key IS NULL` which is sometimes used to try to keep outer rows – if not present, it's likely an anti-pattern (unless truly meant to filter out non-matches entirely).
    - **SOTA Rewrite:** If the goal was actually to keep all A's and only some B's, move the filter into the JOIN condition (i.e., make it part of the ON clause) so that the join itself only brings in matching B rows of interest while still retaining non-matches as NULLs. For example: `FROM A LEFT JOIN B ON A.id=B.id AND B.value=5`. This way, rows from A that have no B or have B but with a different value will still appear (B columns null), and only those with B.value=5 will have B's data [22] [23]. If the intention was actually to drop A's with no matching B (which means an inner join was intended), then simply use an `INNER JOIN` in the first place or change to an inner join for clarity. The rewrite depends on intent: to preserve outer join behavior correctly, use proper ON clause conditions or a two-step approach (join then apply a condition allowing B nulls explicitly). Another correct rewrite is using a **WHERE (B.value = 5 OR B.value IS NULL)** to explicitly keep the outer rows [21] [24], but that's essentially the same as moving the condition to ON. So the best practice: if you want to filter on B yet keep non-matches, put that filter in ON. If you don't want non-matches, just use inner join.
    - **Safety Check:** Confirm the **intended logic**. If the original query was mistakenly written, the agent needs to guess if the user meant an inner join or a correctly filtered outer join. Metadata can help: If B.value has a NOT NULL constraint and the filter is `B.value=5`, then the original query returns no A that doesn't have a B (because any A without B gives B.value NULL which fails the filter). That suggests an inner join was effectively done. If that matches intent, switching to inner join is fine. If the intent was to truly keep all A's and mark those with B.value=5, then the rewrite with the condition in ON (or OR IS NULL) is needed. The agent could use knowledge of foreign keys: if A->B is a foreign key and B.value=5 is required, maybe they intended only matching B (inner). But if no such constraint or if `LEFT JOIN` was explicitly used, likely they wanted outer behavior. To be

safe, the agent might rewrite to the ON-clause approach, which preserves all A rows. That will change results compared to the original (the original unintentionally dropped A's with no B). However, it's *correcting* a logic bug. The agent might flag this as a potential bug fix. In terms of semantic equivalence, it's not equivalent unless we assume the user intended outer semantics. So there's a judgment call. As a "safety check," the agent should verify if the outer-joined table's filter is not null-suppressing. One heuristic: check if the query returns fewer A rows than A table count (if such stats are available and B coverage isn't total). If yes and they used LEFT JOIN, likely a mistake. The agent must ensure any rewrite doesn't inadvertently drop needed rows or include unintended ones. If converting to inner join, double-check if that changes the multiplicity (shouldn't, since original was effectively inner due to filter). If keeping outer and moving filter to ON, validate that the result now includes previously missing rows (which is probably desired). In summary, use constraints and possibly sample data to infer intent, then apply the appropriate safe rewrite.

48. **Nested CTE Cascade (Excessive Layering without Need)**

   - **Human Error:** Stacking many Common Table Expressions (WITH clauses) or nested subqueries where a flatter single query or fewer steps would suffice. While CTEs improve readability, overusing them (especially if one CTE just SELECTs from another CTE etc.) can confuse the optimizer or lead to repeated computations (in some SQL engines prior to optimizations). This is inefficient if not needed, though syntactically fine. For example, `WITH cte1 AS (SELECT ...), cte2 AS (SELECT ... FROM cte1), cte3 AS (SELECT ... FROM cte2) SELECT ... FROM cte3;` where each just passes data along.
   - **Semantic Trigger:** Multiple CTEs where each one is directly used by the next in a linear fashion, or a CTE that is defined and never actually used [25]. Also, if each CTE doesn't reduce the data (no filters/aggregations until the final), they might be just layering without benefit. This can be spotted by an analyzer seeing that cte2 is `SELECT * FROM cte1` with maybe minor tweaks, etc. Unused CTEs (defined but not referenced) are definitely an anti-pattern (just dead code) [25].
   - **SOTA Rewrite:** Simplify the query by **combining CTE logic** or eliminating unnecessary ones. If cte2 just filters or adds a column to cte1, possibly merge that logic into cte1 or skip cte2 by applying the filter in the final query. Modern optimizers like PostgreSQL 12+ inline CTEs (unless marked MATERIALIZED), but not all engines do – and even then, readability may suffer. The rewrite could be to collapse the nested CTEs into one CTE or even directly into the final query. Alternatively, use **subqueries or derived tables** where appropriate if the CTE layering was solely for scoping. The key is to remove indirection that doesn't add semantic value. This might also allow the optimizer to consider reordering operations that were fixed by CTE boundaries (especially in older versions where CTE was an optimization fence).
   - **Safety Check:** Ensure that by merging or removing CTEs, the **result stays the same**. CTEs can sometimes materialize results (especially in older PG or if the CTE is recursive or manually materialized). If the query relied on a CTE to execute a certain way (like forcing an intermediate result for performance or correctness), merging might change behavior. For example, if the CTE was selecting from a volatile function or using `LIMIT` without order, merging could alter which rows are seen (though that would be a questionable query). The agent should check if any CTE uses features like `LIMIT/OFFSET`, window functions, or if it is recursive – those are cases where you shouldn't blindly flatten. Also, if an **identical CTE is used multiple times** in the final query, leaving it as a CTE might avoid duplicate computation (depending on engine). If we inline it, we'd repeat the logic. The

agent can see usage count: if a CTE is referenced twice, it might be better to keep it (or ensure the optimizer will compute once anyway). Ensure no naming collisions when merging (column names etc.). Also, confirm that unused CTEs can be dropped – they have no effect on output, so that's safe by definition. The main risk is if the user intended the CTE to be evaluated once for performance reasons (like a heavy subquery used twice, relying on PG's old behavior to not recompute it). Modern approach would be to use TEMP tables or ensure the optimizer's behavior. Since this is an agent rewriting for optimization, we assume dropping it is beneficial if truly unneeded. So verify none of the removed CTEs had side-effects (they typically don't, as CTEs are just subqueries). For correctness, merged queries should be logically equivalent – the agent can test on a small dataset if possible to ensure results match, but in principle the relational algebra is just fused, which is fine.

49. **Union vs Union All (Unneeded Duplicate Elimination)**

    - **Human Error:** Using `UNION` (which eliminates duplicates across combined results) when duplicates are not a concern or cannot occur, instead of `UNION ALL` which is more efficient. This is a performance anti-pattern: `UNION` sorts/merges results to drop dups which might be unnecessary overhead [26] [27]. Example: combining results from two disjoint categories or two different source tables that have no overlapping rows – a UNION ALL would suffice.
    - **Semantic Trigger:** The presence of the `UNION` keyword where an examination suggests the result sets are inherently distinct or the user did not explicitly need unique results. If each subquery selects from completely separate tables or groups (e.g., "active users" vs "inactive users" from the same table with a condition), any overlap means a user is both active and inactive, which is impossible by definition (assuming that's mutually exclusive). Or if they *did* need unique, but one subquery is already ensuring uniqueness (say using GROUP BY or distinct itself), then an extra UNION distinct is redundant. Essentially, check if a `UNION` could be safely replaced by `UNION ALL`.
    - **SOTA Rewrite:** Replace `UNION` with `UNION ALL` **if** duplicate results are not logically possible or not a concern [26]. This avoids the duplicate-elimination step. For example, `SELECT ... FROM A UNION SELECT ... FROM B` can become `UNION ALL` if A and B data sets don't overlap or you don't mind duplicates. If the intention was actually to remove duplicates, consider whether those duplicates could be better handled earlier (like via a DISTINCT in one part, or if they come from a join issue, fix that). But if truly needed, then you must keep `UNION`. Modern systems often can optimize UNION ALL significantly better, especially if parallelizing or if the inputs are large.
    - **Safety Check:** Only do this if **equivalence is guaranteed**. The agent must check that the two (or more) unioned subqueries cannot produce the same row. This can be done by examining their source tables and conditions. For instance, if each subquery has a predicate on a column that is mutually exclusive (like `status = 'active'` vs `status = 'inactive'`), or if they select entirely different columns or different tables that share no data, then duplicates can't happen [28]. If subqueries are from the same table but with different filters, verify those filters are disjoint (using check constraints, boolean logic, etc.). If there is any doubt, switching to UNION ALL could introduce duplicate rows in results where previously they were merged – that would change query semantics. So the agent should be conservative: if the user didn't explicitly ask for unique results but just used UNION out of habit, it might be safe to assume UNION ALL is okay. But sometimes users use UNION as a quick way to dedupe when combining data – losing that might be unexpected. One approach: if the result is later aggregated or distinct anyway, duplicates don't matter. Or if performance is critical and duplicates minimal, maybe still preserve

correctness strictly. The agent could attempt to prove disjointness by using information_schema (e.g., a constraint that a column can only have certain values per partition, etc.). If not provable, it might leave it or at least warn. In summary, ensure no logical duplicate possibility or that duplicates are acceptable. Formal guarantee would come from integrity constraints or obvious logic. For example, union of two queries selecting different literal tags (like one query selects `'East' AS region, ...` and another `'West' AS region, ...`) clearly yields disjoint outputs because of that extra column – in such cases, safe to use ALL. Also, note that `UNION` vs `UNION ALL` can affect performance but not which rows (except duplicates), so the only semantic difference is duplicates. If duplicates did occur in the original and were removed, adding them back is a change in semantics. Only proceed if sure that case doesn't arise or was unintended.

50. **Partial Key Join (Incomplete Join Conditions)**

    - **Human Error:** Joining tables on only part of a composite key or insufficient conditions, leading to an inadvertent **many-to-many merge**. This often yields duplicate or multiplied rows. For example, table `OrderDetails` has composite key (order_id, product_id), but a join to Products is done only on product_id, ignoring order_id, thereby matching a product to all orders that have that product, instead of just its specific order – resulting in duplicate order rows for each shared product. This is logically incorrect or at least inefficient because of the expanded result.
    - **Semantic Trigger:** A join between two tables that share multiple logical relationship columns, where the join condition only uses some of them. If a foreign key exists on multiple columns but the query only joins on one, it's a clear indicator. Also, if right after such a join the query uses DISTINCT or GROUP BY to fix duplication, that's a sign the join was too broad. We might see something like `ON a.id = b.id` while the real key should be (id,type) = (id,type). The symptom is often unexpected high row counts (join explosion).
    - **SOTA Rewrite:** Include **all necessary columns in the join condition** to represent the true relationship. Using schema metadata, identify if there is a multi-column primary key or foreign key between the tables and use the full key. For example, change `... FROM OrderDetails od JOIN Products p ON od.product_id = p.id ...` to `JOIN Products p ON od.product_id = p.id AND od.tenant = p.tenant` if both tables are multi-tenant and require tenant for uniqueness. Essentially, restore the lost join predicate. If the join was actually meant to be a cross-product (rarely), then the DISTINCT or aggregate after it was the only thing preventing explosion – in that case, reconsider the query logic entirely. Another approach: if truly only part of the key is available or relevant, perhaps the data model is denormalized and the join duplication was intended for something like matching by category rather than exact key – if so, treat it as a different query altogether. But generally, assume it was a mistake: the fix is to join on full key. This eliminates duplicate rows and likely allows the optimizer to use indexes on all key parts.
    - **Safety Check:** This is often a correctness fix, not just optimization. Adding join conditions can *remove* rows that were previously in the result (the duplicates or the improperly joined ones). The agent should verify via foreign key info: does a foreign key exist that spans those columns? If yes, using all columns is definitely correct and will simply remove the unintended multiplicative effect [11]. If no declared foreign key but logically it should be, try to infer from indexes or naming conventions. To be safe, compare the count of distinct key values before and after. For instance, ensure that for each primary key of the left table, the join only returns as many rows as the child table has with that key (if that

was the intention). If previously it was multiplying beyond that, that was wrong or redundant. The agent should also check if any duplicates were being eliminated by a subsequent DISTINCT – if yes, the rewrite likely changes that step (distinct may no longer be needed). This falls under equivalence checking: if we drop DISTINCT after fixing the join, the final output should remain the same (just achieved in a better way). So ideally, remove both the incomplete join and the band-aid DISTINCT together, verifying that final results match on some sample. If the user intended the broad join (maybe to combine categories as mentioned), then adding a condition might drop intended combos – but that scenario is rare and usually handled differently (like separate query). Metadata like **referential constraints, unique indexes** are key to identify the proper join columns. The agent uses them to decide on the fix and to ensure the join becomes one-to-one or one-to-many as intended, not many-to-many. This both improves performance and yields correct result sets without needing distinct. Always double-check: if any result rows vanish or multiply in a test dataset by adding the condition, reevaluate if it's truly the intended logic.

51. **NOT IN vs NOT EXISTS (Null-Sensitive Anti-Join)**

    - **Human Error:** Using `NOT IN (subquery)` without accounting for `NULL` values in the subquery results. In SQL, if the subquery returns even one `NULL`, a `NOT IN` condition will yield FALSE (or UNKNOWN) for all rows, potentially returning no results [29] [30]. For example, `SELECT * FROM customers WHERE id NOT IN (SELECT cust_id FROM orders);` will return an empty set if any order has `cust_id = NULL`. Often, the intent is to find customers with no orders, but if `NULL` sneaks into the orders.cust_id (or the subquery returns no rows), the logic fails.
    - **Semantic Trigger:** A `WHERE ... NOT IN (SELECT ...)` clause, especially if the inner query could produce nulls (e.g., the inner column is not a primary key or not declared NOT NULL). If there's no additional `AND inner_col IS NOT NULL` in the subquery, this is an anti-pattern to watch. It's also triggered when comparing multi-column lists with NOT IN (even one null in a composite yields issues).
    - **SOTA Rewrite:** Use `NOT EXISTS` with a correlated subquery, which properly handles nulls by design [31]. For example: `SELECT * FROM customers c WHERE NOT EXISTS (SELECT 1 FROM orders o WHERE o.cust_id = c.id);` will correctly return customers without orders, regardless of nulls in orders.cust_id. Alternatively, add an explicit `WHERE cust_id IS NOT NULL` in the subquery to guard the NOT IN (though this is sometimes overlooked or misordered). The `NOT EXISTS` approach is clearer and not vulnerable to the null issue [32]. Modern optimizers treat NOT IN and NOT EXISTS similarly performance-wise (when nulls are handled), but correctness is the main concern.
    - **Safety Check:** Changing `NOT IN` to `NOT EXISTS` will change results only in the case where the `NOT IN` was mistakenly returning nothing due to a null. We must ensure that the **new query does what the user intended**, which is usually to exclude items present in the subquery. If the inner subquery's column is known to be NOT NULL (check information_schema for that column's nullability or if it's a primary key/foreign key [33]), then `NOT IN` and `NOT EXISTS` are logically equivalent in outcome. In that safe scenario (no null possible), the rewrite is a performance/style improvement but doesn't change result. If the column can be null, then the original query's result was likely *wrong* or at least unexpected (possibly empty). The agent should verify if any nulls actually occur (maybe by looking at column statistics or constraints). If none occur in practice, both queries are same; if they do, the `NOT EXISTS` version will return more rows (the presumably correct set). This is a beneficial change but technically alters semantics to be correct. Ensure that no other subtle differences exist: if the subquery could return

duplicate values, `NOT IN` doesn't care (it checks each candidate against a set), `NOT EXISTS` will also properly handle it (the existence check is true if at least one match). They're semantically the same except for the null issue. So the main check is inner subquery nullability. Also, check that the subquery in NOT EXISTS properly correlates on the right columns. The agent should identify the relationship between outer and inner query (commonly a foreign key relationship). For performance, ensure the subquery is indexed on that join column (or that the optimizer can do an anti-join efficiently). In summary, verify inner select column is either non-null (making rewrite optional but still good) or possibly null (making rewrite fix semantics). If the agent cannot be sure, opting for `NOT EXISTS` + a comment might be the safest route as it always preserves the intended "none of these values" logic.

Part 2: The Metadata RAG Framework

A robust AI-based SQL optimization agent must leverage database **metadata** and **statistics** to ensure that semantic rewrites (like those in Part 1) are safe and effective. This involves a *Retrieval-Augmented Generation (RAG)* approach where the agent retrieves schema info and runtime stats to ground the LLM's suggestions in reality. Key components include:

- **Information Schema Requirements:** The agent should query the DBMS's information schema (or data dictionary) for critical metadata about tables, columns, and constraints. Specifically, the following info is required to validate the rewrites in Part 1:
- **Primary and Unique Keys:** From `information_schema.table_constraints` and `information_schema.key_column_usage`, gather which columns (or sets) are primary keys or have UNIQUE constraints [5]. This is essential for "safety checks" – e.g., knowing a join column is unique on one side guarantees no duplicate multiplication when rewriting a subquery to a join, or when removing DISTINCT because a key ensures uniqueness.
- **Foreign Keys:** From `information_schema.referential_constraints` (and key_column_usage to map the columns), identify relationships between tables. This helps determine one-to-many vs one-to-one joins and is crucial for deciding if a rewrite is semantically safe. For example, a foreign key from orders(cust_id) to customers(id) tells the agent that replacing an EXISTS subquery with a join won't duplicate customers (each order links to exactly one customer). Foreign key metadata was used by Snowflake's optimizer for join elimination [5] and our agent similarly uses it to verify rewriting a join + DISTINCT to a simpler join is valid.
- **Column Data Types and Nullability:** From `information_schema.columns`, retrieve each column's type and whether it can be NULL. Data type informs how to rewrite expressions (e.g., avoid rounding issues in numeric ops, or know collation for strings). Nullability is critical for patterns like NOT IN vs NOT EXISTS – if a column is NOT NULL, we know `NULL` values won't appear [31]. It also matters for join logic (nullable FKs might mean not all parents exist) and for function rewrites (we must preserve null handling).
- **Check Constraints and Enumerations:** If available, any CHECK constraints that restrict column values (e.g., a status can only be 'Active' or 'Inactive') help the agent reason about mutual exclusivity (useful for determining if OR conditions or UNION branches are disjoint). Also, knowledge of distribution (like one table covers all possible values of a domain, etc.) can guide safe removal of duplicates or use of UNION ALL.
- **Indexes (for cost estimation):** Though performance is secondary to correctness here, knowing indexes (via system catalogs or `information_schema.statistics` in MySQL, or `pg_indexes` in Postgres) helps predict if a rewrite will likely improve performance (e.g., if an index exists to support a new join or predicate form). The agent might favor a rewrite that enables index usage (like moving a function off a column) especially if it sees an index on that column.

- **Table Row Counts and Histograms:** While not in standard information_schema, many systems have tables like `pg_class` (Postgres) or `INFORMATION_SCHEMA.TABLES` (which often has row count estimates) or related statistics views. These numbers tell how big each table is, which is important context.

- **Statistical Metadata & Cardinality:** Beyond static schema, the agent should retrieve *statistical metadata* – estimates or actual values about data distribution and table sizes:

- **Cardinality Estimates:** Use the query planner's estimates or database statistics to get row counts for tables and intermediate joins. For example, Postgres's `pg_stats` or MySQL's `information_schema.TABLES` (with `TABLE_ROWS`) provide rough row counts. This data can prevent LLM hallucinations about which table is "small" or "large". If a rewrite is only beneficial when a table is small (or vice versa), having actual cardinality guides the decision. Modern research (LLM-R², E^3-Rewrite) includes cardinality and cost in the prompt to the LLM [34], and they observe it improves decisions.
- **EXPLAIN Plans:** The agent can execute `EXPLAIN [ANALYZE]` on the original query (or parts of it) to see how the optimizer approaches it and get estimated rows for each step. This real-time plan data can be injected into the LLM's context. For instance, if the plan shows a nested loop expected to iterate millions of times, the LLM can focus on that as a problem area. By providing the cost and cardinality of operations, the LLM's reasoning is grounded: it won't, say, recommend an index that doesn't help a low-selectivity predicate if it sees the high row count.
- **Cardinality Estimation Hallucinations:** Without actual data, an LLM might wrongly assume the impact of an optimization. For example, it might suggest pushing a predicate or using an index assuming a table is large, but if the table has 10 rows, the effort is pointless. Conversely, it might ignore a costly join reordering because it "imagines" tables are small. By feeding the LLM with real cardinalities and maybe distinct value counts, we avoid these hallucinations. In practice, an LLM given no stats might incorrectly deem two queries equivalent in performance or might pick a suboptimal rewrite. Real-time info like "Table A has 10M rows, Table B has 100 rows" helps it choose the right join order or realize why a subquery approach might be costly.

- **Explainability and Confidence:** The agent can use metadata to double-check the LLM's suggestion. For instance, if the LLM proposes removing a DISTINCT, the agent can verify using metadata whether duplicates are actually possible (by checking key constraints). If the LLM suggests an index that already exists, the agent sees that in metadata and avoids redundant advice. Essentially, the metadata acts as a fact-checker for the LLM's ideas, preventing nonsensical or inapplicable rewrites.

- **Real-time Data Injection:** To systematically incorporate this, the agent employs a retrieval step before generation:

- *Schema Retrieval:* Before rewriting, the agent gathers all relevant schema info for tables involved in the query (keys, FKs, etc.). This forms part of the LLM's context or system prompt.
- *Statistics Retrieval:* The agent may run `EXPLAIN` on the original query or parts of it to get cost and row estimates, injecting highlights of that (e.g., "The optimizer expects 5 million rows from joining A and B, using index X with selectivity 10%").
- *Prompt Construction:* With these facts, the prompt might say: *"Table A has 5,000,000 rows, Table B has 200. A.id is primary key; B.a_id is foreign key referencing A.id (one-to-many). Current plan: Nested Loop, B as outer. Consider refactoring."* This grounding ensures the LLM's rewrite reasoning is anchored to actual database characteristics, greatly reducing blind spots.

In summary, the RAG framework for our SQL optimizer involves marrying the LLM's generative suggestions with **authoritative metadata** from the database. This guards against logical errors (via constraints info) and performance pitfalls (via stats). Notably, research prototypes like the CMU verified rewrite system explicitly feed schema & stats to the LLM [35], and LLM-based optimizers use execution feedback to refine their output [36]. Our agent will similarly **"retrieve then reason"**, ensuring each rewrite it proposes passes a metadata-based sanity check for correctness and efficiency.

Cardinality Estimation Hallucinations and Mitigation

Large Language Models lack an internal model of the actual data volumes, which can lead to "cardinality hallucinations" – false assumptions about how many rows a table or join produces. This can manifest in suggestions that are theoretically good but practically irrelevant or even harmful: - **Example Hallucination:** An LLM might rewrite a query to use a `HASH JOIN` by rewriting SQL, thinking two tables are large and of similar size, when in reality one table is tiny (and a nested loop or index join was fine). Or it might suggest a multi-step CTE to reduce data early, assuming a huge intermediate result, whereas the actual data is small and the original one-liner was fine. Without real stats, the LLM is guessing.

To combat this: - **Inject Real Cardinalities:** As mentioned, provide actual row counts or size ratios in the prompt. For instance, if the LLM sees "Table Events has ~10^8 rows, Table Users has ~10^4 rows", it will reason differently than if both were unknown. It might then rightly suggest filtering Events before join, or using a semi-join, etc. LLM-R^2 (2024) explicitly notes that estimated cardinality and cost are important context and includes them when prompting the model [34]. - **Feedback Loop with EXPLAIN:** After the LLM proposes a rewritten query, the agent can **run EXPLAIN on the new query** to see if the plan is indeed better (lower cost, or using indexes, etc.). If the plan is worse or the new cardinality estimates explode, the agent might reject or adjust the suggestion. This creates a validation loop where the model's output is checked against reality and potentially fed back for refinement. - **Data Sampling for Equivalence:** Sometimes, to ensure equivalence and gauge performance, the system can run both original and rewritten queries on a small sample of the data or synthetic dataset [35]. By comparing results, we verify correctness; by comparing timing or plan cost, we verify performance. This guards against cardinality estimation errors by actually observing them on data. While not always feasible on full data, even a sample can catch big differences.

The key is that **an AI-based rewrite system must be "execution-aware"** [36]. By grounding the LLM in execution data (cardinality, cost), we drastically reduce the risk of hallucinated optimizations. Real-time stats turn the LLM from a theoretical advisor into a context-tuned one: As the E^3-Rewrite framework describes, incorporating execution plans (with row counts and bottlenecks) into the prompt gives the model awareness of performance issues [36]. Likewise, our agent will use the information_schema and planner estimates as a factual knowledge base to drive safe and sensible rewrites.

Part 3: Comparative SOTA Analysis

In this section, we compare how state-of-the-art systems and research prototypes handle semantic SQL rewrites and validation. We focus on: (a) **Equivalence Checking** – ensuring the optimized query returns the same results as the original, and (b) the **Validation Loop** architecture – how the system iteratively verifies and refines the LLM (or rule-based) outputs. We consider four examples: **R-Bot (VLDB 2024/25)**,

EverSQL (commercial), **Snowflake's approach (e.g., Cortex/optimizer)**, and **SQLGenie (2025 Amazon)**.

- **R-Bot (LLM-based Query Rewrite System, PVLDB 2024):**
  Equivalence Checking: R-Bot addresses correctness by leveraging **established rewrite rules** and multi-step verification rather than letting the LLM freely generate SQL. It retrieves a knowledge base of proven transformation rules (from query optimization literature/engines) and has the LLM apply these rules, so each applied rule is known to produce an equivalent query [37]. This significantly reduces the chance of non-equivalent output, since the LLM is effectively constrained to valid rule patterns. Moreover, R-Bot uses a *self-reflection* mechanism: after proposing a rewrite, the system can reflect (likely by analyzing if the rule was applicable and correctly applied) before finalizing [38]. The paper doesn't explicitly state a formal equivalence prover in use, but by sticking to rules, they implicitly rely on the database optimizer's algebraic equivalences. Any complex sequence of rules could still go awry, so presumably R-Bot might test the rewritten query on a sample or at least ensure no syntax/semantic errors. They mention avoiding hallucinations by guiding the LLM with retrieved evidence and iterative steps [39], which implies each intermediate query is checked against the evidence (the rules documentation). If a rewrite rule's preconditions aren't met (which could be like our "safety checks"), R-Bot's retrieval step would not surface that rule, so the LLM wouldn't use it. This acts as an implicit equivalence guard – only correct rewrites are even considered.
  Validation Harness: R-Bot's architecture is a **multi-step pipeline** [39]:
- **Rewrite Evidence Preparation:** It builds a knowledge base of rewrite rules and examples from multiple sources (like textbooks, optimizer rules, previous queries).
- **Hybrid Retrieval:** Given an incoming query, it retrieves the most relevant rules (e.g., if the query has a subquery, fetch rules about subquery flattening, etc.) [40].
- **LLM Rewrite with Self-Reflection:** The LLM is prompted in a step-by-step manner: pick a rule, apply it, produce a new SQL, then reflect if further improvement is possible [38]. The *validation loop* here is primarily internal: after each rule application, the system could parse the SQL to ensure it's equivalent in structure (they might, for example, verify that the new query, when parsed to a tree, is the same as applying that rule to the original query tree).

- **Iterative Improvement:** R-Bot iterates until no more rules apply or a stopping criterion (maybe a certain cost improvement threshold or rule count limit) is reached [41]. Throughout, if the LLM ever proposes something not supported by evidence, that's likely caught as hallucination (since it won't match a known rule pattern). In deployment, they likely also tested the final output query on the database to ensure performance gain and correctness. Since R-Bot was deployed at Huawei, one can assume they had a final verification step (perhaps running both queries on a test database to confirm equal results, especially if critical) – though the paper's main focus was on performance. Summarily, R-Bot's harness trusts *formalized rules* for equivalence and uses an iterative LLM-agents approach to refine rewrites, minimizing human intervention while maintaining correctness by design [37]. It's a novel marriage of rule-based and LLM-based methods.

- **EverSQL (AI-Powered Query Optimizer, 2025):**
  Equivalence Checking: EverSQL is a production tool that suggests optimized SQL and index recommendations. To ensure the rewritten query is equivalent, EverSQL relies on **conservative, well-known transformations** and **user validation**. It doesn't use an LLM to generate arbitrary SQL; instead, it applies a set of optimization patterns (likely coded by experts). For example, it might replace a join+GROUP BY with an EXISTS subquery or vice-versa [19], or suggest moving a function out of a WHERE clause [19]. These patterns are standard and preserve results by definition (under the right conditions). EverSQL's web interface often provides explanations

("Explaining the magic") describing *what was changed and why* [42] – implicitly assuring the user the result set is the same. The *actual* equivalence is not formally proven by EverSQL internally (there's no mention of using solvers); instead, the trust comes from the simplicity of the transformations and the assumption that the user will test critical queries. The tool is non-intrusive (doesn't run the query on your data for validation) [43] [44]. So equivalence checking is primarily via static analysis: EverSQL will not drop a condition or join; it preserves filters and joins while reordering or refactoring them. If a rewrite could risk a behavioral change, it likely either avoids it or provides it only as an option with explanation. For instance, it wouldn't suggest removing DISTINCT unless sure by schema analysis it's redundant. Thus, EverSQL's approach to equivalence is **rule-based and heuristic**, using integrity metadata where possible to justify changes (e.g., "we detected this join has a one-to-many relationship, so we can safely use EXISTS instead of GROUP BY distinct" – not a direct quote, but an expected rationale).
Validation Loop: EverSQL's workflow is basically one-shot suggestions rather than an interactive loop. The user inputs a query, and EverSQL's engine analyzes the SQL parse tree and the schema (the user may provide schema or EverSQL can parse CREATE statements). It then generates recommendations: rewritten query and indexes. The *human* is in the loop to review and test. Because it doesn't execute queries itself (for privacy/security) [43], EverSQL can't do an automatic before-and-after result check. Instead, the "validation harness" is their internal QA plus the user's acceptance. However, EverSQL does emphasize monitoring after deployment [45]. It includes a continuous performance monitor that could catch if a suggested change caused an issue. But that's performance validation, not equivalence. Equivalence is assumed given the nature of changes. In summary, EverSQL handles correctness by **limiting itself to proven-safe transformations** and relies on the user to run the optimized query to confirm it yields the same results (since it's often integrated in development workflows). Its architecture is not an online agent like R-Bot; it's more of a batch advisor with a static set of optimizations, thus no iterative loop to speak of – just analyze -> output recommendations. The explanations it gives serve as a partial validation, describing why the new query is the same logically but faster (e.g., "replacing the INNER JOIN and GROUP BY with EXISTS subquery… allowed avoiding a large intermediate result without changing the outcome" [46]). This transparency helps the user trust equivalence.

- **Snowflake (Cortex & Native Optimizer):**
Snowflake's "Cortex" is an initiative to integrate LLMs into the Snowflake Data Cloud (including user-defined functions with LLMs, etc.), but relevant to query optimization is Snowflake's **built-in optimizer**. Snowflake doesn't (yet) have an external LLM rewriting SQL for you; rather, its **query optimizer internally rewrites queries** for performance. *Equivalence Checking:* In Snowflake's optimizer, any rewrite it does is governed by strict rules and sometimes relies on **constraints declared by the user**. For instance, Snowflake can do **join elimination** (remove redundant joins) if it knows a foreign key exists and the join doesn't contribute columns needed [5]. It marks constraints as RELY for this purpose (user asserts them as true) [47] [5]. The optimizer will only remove that join if it's certain it doesn't affect results [5]. Equivalence is guaranteed by relational algebra correctness (they test those rules extensively). When Snowflake's engine rewrites a query (say flattening subqueries, pushing predicates, etc.), it does so within the C++ optimizer code, and there's no doubt of equivalence because it's effectively doing algebraic transformations. In terms of exposing this, Snowflake's UI will show the final executed query plan, which might look different than the user query [48], but users trust the DBMS to preserve semantics. For LLM or "Cortex" involvement, one scenario is using an LLM to suggest a better SQL to a user (like a separate tool fine-tuned on Snowflake). In that case, the equivalence check might come down to simply running both queries on a sample and comparing results, or leveraging Snowflake's optimizer to confirm both yield same plan or results. *Validation Loop:* Snowflake's native approach doesn't require an external loop – it's all inside the optimizer which cost-based chooses the best plan (which might be an implicitly rewritten form). However, if we imagine a cloud AI

assistant (like in the Reddit example, someone fine-tuning an LLM on Snowflake queries [49]), the validation loop would involve:

1. The LLM suggests a rewritten SQL.
2. The system executes both original and new (maybe on a small subset or full if feasible) to ensure identical results and compares query profiles. In the Snowflake context, since the engine is robust, one could also rely on comparing the query plans – if the logical plans are equivalent (same result set derivation), Snowflake likely can tell via its plan tree. Snowflake's team of PhDs (as mentioned in the Reddit comments) tests any new optimization extensively to ensure no change in results [50] [51].
3. If the new query is indeed faster and correct, it could be offered to the user or auto-applied. Snowflake's "Project Cortex" is more about using ML for predicting result sizes or indexes rather than rewriting SQL text. But interestingly, Snowflake's recent capabilities like **Search Optimization Service** and **Materialized Views** automatically rewrite queries under-the-hood. They validate equivalence by only using them when certain (for example, a materialized view will only service a query if it's guaranteed to have the same result given filters etc.). Those guarantees come from **optimizer rules** and metadata. So Snowflake's approach can be summarized: *strict rule-based equivalence, no guesswork.* They either know it's equivalent (by constraint or algebra) or they don't do it. The "harness" is built-in testing and the cost-based decision process. For an LLM agent outside Snowflake, one would likely integrate with Snowflake by retrieving constraints (like the Medium article by Hoffa shows adding constraints and using them [52] [47]) and maybe using Snowflake's `EXPLAIN` to validate. In short, Snowflake's state-of-the-art approach to safe optimization is **to use the optimizer itself as the validator** – any externally suggested rewrite can be checked by seeing if Snowflake's optimizer already does it or by running an actual result comparison. (Snowflake's ability to quickly clone databases or use zero-copy clones could facilitate test execution for an agent, but that's beyond our scope.)

- **SQLGenie (LLM-based NL→SQL system, Amazon 2025):**
SQLGenie is focused on *natural language to SQL generation*, but it introduces relevant ideas for reliability and efficiency. *Equivalence Checking:* In SQLGenie's case, the "equivalence" is between the user's question intent and the generated SQL's results. They ensure reliability by a **Feedback Augmentation** component that filters out incorrect LLM outputs [53]. They likely run the generated SQL on sample databases or use an ensemble of LLM agents to cross-verify (one generates, another critiques). For optimization (efficiency), SQLGenie's Table Onboarder collects metadata like foreign keys, indices, partitioning [53] [54] so that the generated SQL not only is correct but also performant. They implicitly do an equivalence check by testing queries on data: execution accuracy on benchmarks (e.g., did the query return the expected answer for NL questions) is one metric [55]. If an LLM suggests an alternative SQL (for the same NL query), they can compare the results with the ground truth answer – a sort of result-based verification. For an optimization context, SQLGenie doesn't directly rewrite SQL given as input, but we can extrapolate: a similar system could generate multiple candidate rewrites and then test them on the database to see if results match and which is fastest. This is akin to brute-force validation: run queries and compare outputs [56]. SQLGenie's multiple-agent approach (they mention leveraging multiple LLM agents for complex SQL and storing verified examples) hints at a *loop*

where an agent generates SQL, another (or the same with a different prompt) verifies it, and the system learns from that [55] . *Validation Loop:* The architecture likely involves:

1. **Table Onboarding:** Pre-compute metadata (like stats, keys) so that any generation knows the schema context (similar to our Part 2, they feed the LLM schema knowledge to avoid invalid joins or filters) [53] .
2. **SQL Generation:** LLM proposes a query.
3. **Verification & Refinement:** They use either execution (running the query on sample data to see if it answers the NL question correctly) or a second LLM pass that checks the SQL against expected forms or known correct outputs. They mention filtering correct query-SQL pairs and storing verified examples [55] , which implies they generate many and test them, keeping those that were correct. This is essentially a *result-based equivalence check*: if the output matches the known answer, the query is considered equivalent to the spec.
4. **Efficiency check:** Since SQLGenie also cares about performance, the Table Onboarder's index/partition suggestions ensure the generated SQL uses those (the LLM might be guided to include certain hints or structures for speed). If multiple equivalent queries exist, they might prefer the one that, given the metadata, would run faster (though how the LLM decides that is unclear – possibly via few-shot examples showing that certain patterns are faster).
5. **Continuous Learning:** The verified examples are stored, meaning the system improves and possibly uses those as future prompts or fine-tuning (so over time it is less likely to generate an incorrect or slow query because it has seen many correct ones). While SQLGenie is not an optimizer for given SQL, its methodology highlights how an LLM system can be made reliable: **multi-agent prompting, execution feedback, and a knowledge base of proven cases**. This aligns with general trends in LLM-based DB tools: use the DB to check the DB. For our optimization agent, a similar harness would be: generate rewrite → run on small data to check equivalence (like a differential test) [56] → if not equal, adjust or discard → possibly try a different approach, etc. Also, SQLGenie's use of foreign key mapping [54] is analogous to our metadata usage: by knowing relationships, the system avoids making joins that produce wrong cardinalities or missing necessary filters (so its outputs are more likely correct and efficient).

To sum up, current SOTA solutions ensure correctness and performance via a combination of **rule-based guarantees, retrieved evidence, and runtime verification**: - R-Bot uses retrieval-augmented LLM with formal rewrite rules and iterative self-checking [39] . - EverSQL uses a static rule engine with implicit equivalence (no risky rewrites) and relies on human oversight [19] [46] . - Snowflake's optimizer uses internal rule-based rewrites gated by constraints, essentially a closed-loop where the optimizer's cost model and rules validate the changes (and it surfaces the final plan for transparency) [48] . - SQLGenie shows the value of multiple LLM agents and real query execution to validate and tune queries, storing those results for future robustness [55] .

Validation Loop Architectures in Summary: Many systems employ a *two-phase approach*: generation of candidate rewrites followed by verification either by a formal method or actual execution. For example, one design (as seen in a CMU project [35]) is: 1. Feed schema & stats to LLM, get a rewritten query. 2. Use a **semantic equivalence prover** (like the Cosette or QED tool) to check query equivalence or generate counterexamples. 3. If formal proof is hard, generate some test data, run both queries, and compare results [57] . 4. If mismatch, either adjust the prompt (LLM self-reflection) or try another strategy; if match, then proceed to performance evaluation. 5. Performance check via EXPLAIN or running on a small slice to compare execution time or cost. 6. If performance is indeed better (taking into account the DB's own optimizer might already have done something similar), then deliver the new query.

This loop ensures the new query **equals** the old (step 2-3) and is **faster** (step 5) before suggesting it. Some research (like E³-Rewrite) even bakes this into a reinforcement learning reward, so the model is trained to only produce equivalent and faster rewrites, effectively learning the validation internally [58] [59].

In conclusion, SOTA systems combine **LLM strengths (natural language and pattern generalization)** with **traditional database guarantees (rules, integrity constraints, cost models)**. Equivalence checking ranges from using known safe rule-sets (preventing any non-equivalent output) to formal provers and runtime tests [60] [61]. The validation harness often involves iterative refinement: an initial rewrite, followed by either automated proofs or actual execution comparisons, and potentially feeding that feedback to the model (either via prompt or training). All of this aligns with a core principle: never trust a single LLM output blindly – always verify with the database itself or a formal method before assuming an optimization is valid. This way, we get the best of both worlds: novel query rewrites suggested by LLM's broad knowledge, and rock-solid assurance from database theory and practice that the final query is correct and superior.

---

[1] Advanced Snowflake SQL – Lesson 2: Subqueries and Common Table Expressions (CTEs) | BitKraft
https://bitkraft.in/articles/advanced-snowflake-sql-lesson-2-subqueries-and-common-table-expressions-ctes/

[2] [3] [4] [6] [7] [8] [12] [13] [14] [15] [16] [17] [18] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] Mastering SQL: How to detect and avoid 34+ Common SQL Antipatterns
https://sonra.io/mastering-sql-how-to-detect-and-avoid-34-common-sql-antipatterns/

[5] [11] [47] [52] How to help Snowflake eliminate your redundant joins with constraints | by Felipe Hoffa | Medium
https://hoffa.medium.com/how-to-help-snowflake-eliminate-your-redundant-joins-with-constraints-f6c696264114

[9] [10] [19] [42] [43] [44] [45] [46] EverSQL Query Optimizer: My Secret Weapon for Peak AI Performance
https://skywork.ai/skypage/en/everSQL-query-optimizer-ai-performance/1977568012666990592

[34] VLDB_24_03_revision
https://www.vldb.org/pvldb/vol18/p53-yuan.pdf

[35] [57] 15799 Final Presentation
https://15799.courses.cs.cmu.edu/spring2025/files/final/llm-query-rewrite.pdf

[36] [58] [59] E3-Rewrite: Learning to Rewrite SQL for Executability, Equivalence, and Efficiency
https://arxiv.org/html/2508.09023v1

[37] [38] [39] [40] R-Bot: An LLM-based Query Rewrite System
https://arxiv.org/pdf/2412.01661

[41] R-Bot: An LLM-Based Query Rewrite System - ResearchGate
https://www.researchgate.net/publication/395559470_R-Bot_An_LLM-Based_Query_Rewrite_System

[48] [49] [50] [51] Using LLM to Optimize Snowflake SQL Queries – Seeking Recommendations and Insights : r/snowflake
https://www.reddit.com/r/snowflake/comments/1h4zm82/using_llm_to_optimize_snowflake_sql_queries/

[53] [55] SQLGenie: A practical LLM based system for reliable and efficient SQL generation - Amazon Science
https://www.amazon.science/publications/sqlgenie-a-practical-llm-based-system-for-reliable-and-efficient-sql-generation

[54] aclanthology.org
https://aclanthology.org/2025.acl-industry.71.pdf

[56] [60] [61] A Survey of SQL Equivalence Verification Methods | by Rebooter.S | Medium
https://medium.com/@Rebooter.S/a-survey-of-sql-equivalence-verification-methods-1b6fdace410c