ChatGPT

Beyond Traditional DB Optimizers: LLM-Driven Query Rewrite Patterns

Introduction

Modern database query optimizers excel at **local, algebraic rewrites** within a single SQL statement – e.g. pushing down filters, reordering joins, simplifying predicates – so long as the transformations are strictly semantics-preserving. These rule-based transformations are powerful but **limited in scope**, and updating them or adding new rules in production is difficult [1]. As a result, traditional optimizers often **"fall down"** when an optimization would require going beyond the given query's text or the engine's guaranteed equivalences. In practice, complex or non-standard queries can lead the optimizer astray and result in suboptimal plans [2]. Recent research shows that **LLM-driven query rewriters** can fill these gaps: by leveraging an LLM's understanding of intent and context, systems like LITHE and R-Bot have achieved dramatic speedups on slow queries compared to native optimizers [3] [4]. Below, we outline key pattern families where **traditional DB optimizers struggle** and how an LLM-based optimizer can shine by handling these cases.

1. Intent-Dependent Rewrites

Sometimes a SQL query is written in a way that is *correct* but not aligned with the user's true intent or optimal usage pattern. A conventional optimizer can't infer the intent behind the query; it only optimizes the given SQL literally. An LLM, however, can **interpret the intent** and suggest a rewritten query that better matches what the user likely meant. These rewrites often change the query structure or semantics slightly to meet the intent (which a DB engine would not do on its own). Key examples include:

- **OFFSET Pagination vs. Keyset Pagination:** Paginating query results by using `OFFSET ... LIMIT ...` is often inefficient for large offsets – the database must scan and discard a potentially huge number of rows. If the intent is user-facing pagination, **keyset pagination** (also called seek pagination) is usually faster. The LLM can rewrite an `ORDER BY ... OFFSET N LIMIT M` pattern into a `WHERE key > last_seen_key LIMIT M` pattern using a **stable ordering column** (usually a unique, monotonic ID or timestamp). This avoids skipping through N rows on every page. A DB optimizer won't do this because it changes the access pattern and requires knowing a stable sort key (often a primary key or unique timestamp) that ensures new inserts don't disrupt pagination order. The LLM must verify the presence of a unique ordering column and that the "next page" will be consistent (e.g. no missing or duplicate entries if new rows arrive). *Detection cues:* use of `OFFSET` with a large offset number in a user-facing query, suggesting the query is for paginating results. The LLM can explain that switching to keyset pagination will make page loads faster and more stable at scale.

- **"Latest Row per Group" Queries:** A common intent is to retrieve the *latest* record for each category or group (e.g. latest order per customer, most recent log entry per device). There are many SQL patterns to achieve this (self-joins, correlated subqueries with `MAX()`, window functions with rank, etc.), but not all are equally efficient. Traditional optimizers do not automatically recognize the intent of "latest per group" – they will faithfully execute whatever

approach the query uses. An LLM-driven optimizer can detect this intent and rewrite the query into an optimal form. For example, if it sees a correlated subquery like `... WHERE t.date = (SELECT MAX(date) FROM T t2 WHERE t2.group_id = t.group_id)`, it can rewrite to use a window function: `ROW_NUMBER() OVER (PARTITION BY group_id ORDER BY date DESC) = 1` (possibly using a `QUALIFY` clause or an equivalent subquery) so that the database only scans once and filters by the row-number. Another approach is using a **JOIN on the MAX** for each group (plus a tie-break on a unique ID if needed). The "best" form can depend on the engine and indexing (some engines support a `DISTINCT ON` or a lateral join for this pattern). The LLM can choose the form that is both semantically correct (tie-breaks properly if two rows have the same latest timestamp) and performant. *Detection cues:* correlated subqueries selecting `MAX()` or self-joins on equality to a sub-select, which indicate an attempt to get max/min per group.

- **Top-N per Group Selection:** Similar to the latest-per-group, sometimes the intent is to get the **top N records for each category** (e.g. the 3 highest-value transactions per account). SQL often requires either a complicated window function or a LATERAL join to express this. Many SQL writers emulate it with subqueries or by retrieving all data and post-filtering in application code. A traditional optimizer won't intuit this intent and has no built-in rule to introduce a LATERAL join or nested loop for "top-N per group." An LLM-based optimizer can detect phrases or patterns like "get N per group" or query shapes that pull all data then filter within groups, and suggest using a more efficient strategy (for instance, using `ROW_NUMBER() OVER (PARTITION BY ... ORDER BY ...)` and then filtering on row_num $\leq$ N, or using a LATERAL subquery that selects N rows from each group). In fact, one known rewrite is to replace a grouped aggregate subquery with a `JOIN LATERAL` (or `CROSS APPLY`) that uses an `ORDER BY ... LIMIT N` inside the subquery for each group [5]. This can drastically reduce the scanned rows if an index on the ordering column exists. The LLM can provide this as a recipe, since most engines won't spontaneously create a lateral join on their own. *Detection cues:* patterns where the query selects from a table, and for each item in another table wants the top N related rows (often indicated by `GROUP BY` combined with limiting or filtering of an inner query).

Why DB Optimizers Won't: These intent-based rewrites often **change the semantics or requirements** of the query slightly. For example, keyset pagination requires a unique ordering and returns different row slices than OFFSET; "latest per group" assumes a definition of "latest" that might need tie-break logic. Database engines won't apply such changes automatically because they cannot be sure it's what the user intended and the transformations aren't guaranteed equivalent without additional assumptions (e.g. uniqueness, stable sort order).

LLM Advantage: An LLM can infer the *intent* (e.g. "the user likely wants pagination, not an expensive offset scan" or "they want one latest record per group") and propose a rewrite that aligns with that intent. It can also ask for or use metadata (like primary key info or timestamp fields) to ensure the rewrite is correct. This allows **intent-aware optimization** that the rigid rule-based optimizer cannot do.

2. Semantics-Changing Tradeoffs for Speed

Traditional optimizers err on the side of *strict correctness* – they will not rewrite a query in a way that changes the result set, even if the change might have negligible impact on analysis or would be acceptable to the user. However, in analytical or non-critical contexts, users might be willing to trade exact correctness for a big performance boost (especially for exploratory queries, dashboards, etc.).

These are scenarios where an LLM-driven approach can suggest **opt-in approximations** or relaxed semantics to gain speed. A few high-impact examples:

- **Exact `COUNT(DISTINCT)` → Approximate Count:** Counting distinct values can be very expensive on large datasets, because it often requires large memory usage or distributed shuffle to gather unique values. Many engines offer approximate algorithms (like HyperLogLog sketches) for COUNT DISTINCT that run much faster with a small error margin. A normal DB optimizer will *never* substitute an approximate result for an exact one on its own – that would violate correctness. But an LLM can recognize when an approximate count might be acceptable (for example, in a **dashboard query or analytic report** where a 1-2% error is tolerable). The rewrite would replace `COUNT(DISTINCT column)` with the engine-specific approximate function (e.g. `APPROX_COUNT_DISTINCT(column)` or using a HyperLogLog UDF). The LLM would need hints about acceptable error tolerance or query context (e.g. not a financial report that needs exact numbers). *Why it helps:* Using a sketch algorithm can speed up distinct counts by orders of magnitude. For instance, in one benchmark counting ~100M unique values, a HyperLogLog approach was **144× faster** than exact count distinct [6]. The LLM can suggest this when `approx_ok = true` in the metadata or if it detects a known pattern like a large telemetry aggregation.

- **Exact Percentiles/Medians → Approximate Quantiles:** Similarly, computing an exact median or percentile (e.g. using `PERCENTILE_CONT` or sorting and taking middle value) is expensive as it may require a full sort of data. Many systems have functions like `APPROX_PERCENTILE` or `percentile_approx()` that use algorithms (t-digest, reservoir sampling, etc.) to estimate quantiles efficiently. A DB optimizer won't swap in an approximate percentile function by itself, since the results aren't exactly the same. But an LLM-driven optimizer can recommend it if the use-case is exploratory or if slight inaccuracy is acceptable. The rewrite would be to call the approximate function (with perhaps a known error bound or compression parameter). The LLM might look for usage of exact median or percentile functions in a query that touches a very large dataset, and if performance is a concern, propose the faster approximate alternative. It would also clarify the trade-off (e.g. "This will be much faster but results are approximate within a small error range"). *Detection cues:* presence of `PERCENTILE_CONT`, `MEDIAN()` or similar in a query on a big table or without limiting, especially in a dashboard context.

- **Heavy Sort or Top-K → Approximate Top-K:** Queries that need the **top K** items by some metric (e.g. top 100 most frequent values, top 10 customers by revenue) typically require sorting or heavy aggregation. If K is much smaller than total rows, there are algorithms to estimate the top K using probabilistic methods or algorithms like Misra-Gries or count-min sketch which avoid sorting all data. A database will not choose a lossy method on its own – it will do the exact sort or exact aggregation. An LLM optimizer could identify a *ranking query* (especially if it's for a UI where perfect accuracy isn't critical) and suggest a transformation such as using a **sampling or sketch-based pre-aggregation** to approximate the top K. For example, it might suggest using a reservoir sample to estimate top categories, or using a summary table updated periodically. This is a more advanced scenario and often would involve more than just SQL (maybe recommending a separate process), but it's within the LLM's capability to **advise a different approach** if the user explicitly allows approximate answers for huge data. *Detection cues:* an ORDER BY with LIMIT on a very large table, especially if the query runs frequently (e.g. "top search queries today" on a site with millions of queries).

- **Duplicate Elimination (UNION vs UNION ALL):** Another semantics-relaxing tweak is replacing `UNION` with `UNION ALL` when de-duplication is not necessary. A `UNION` (without ALL) forces

the query to sort or hash results to eliminate duplicates across the unioned subqueries, which can be very slow if each subquery returns large results. If the user knows that the two subqueries produce disjoint sets or simply doesn't care about the small chance of duplicates, using `UNION ALL` will be much faster (no duplicate check). Traditional optimizers will not make this change because it can change the result if duplicates *do* exist. But an LLM can infer from context or metadata when duplicates are "impossible or unlikely" (for example, union of data from two different time periods or two different categories) [5], and suggest using `UNION ALL`. In cases where duplicates might exist, the LLM could even suggest adding a quick `DISTINCT` later if needed, but often domain knowledge says the union inputs are inherently distinct. *Detection cues:* use of `UNION` in long queries, especially if each branch selects from tables that logically have no overlap (or if performance issues are observed). The LLM would highlight the performance gain at the cost of potential duplicate entries if that assumption is wrong.

Why DB Optimizers Won't: These transformations **change query semantics** – approximations introduce error, and skipping a duplicate elimination could change results. Database engines leave such decisions to the user or require explicit functions (e.g. approximate functions or hints) because they guarantee correctness by default. Without a user directive, they won't return an approximate answer or drop a duplicate removal step.

LLM Advantage: The LLM can act as a smart advisor that **asks the user or relies on configuration** about tolerance for error. It can combine knowledge of algorithms (like HyperLogLog for count distinct) with the user's intent (e.g. "this is a metrics dashboard, not an audit report") to propose high-impact speedups that a normal optimizer cannot. Essentially, the LLM can **"trade correctness for speed"** in a controlled way when it's beneficial and acceptable. This can yield massive performance improvements for analytical workloads [6], as long as the user is informed and consents to the trade-off.

3. Constraint-Dependent Rewrites

Many optimizations are only valid if certain *data constraints or relationships* hold true – for example, if a join is guaranteed one-to-one, or a column is unique, or a foreign key relationship exists. Commercial database optimizers do make use of declared constraints (primary keys, foreign keys, unique constraints) for some optimizations like join elimination or duplicate removal. However, often these constraints are either not declared in the schema or the optimizer is conservative in using them. An LLM-driven optimizer can be more **aggressive in leveraging inferred or provided constraints** to rewrite queries in ways that are not generally safe, but are safe *given the specific database's properties*. Here are some patterns:

- **JOIN followed by DISTINCT → Semi-Join (EXISTS):** Consider a query that joins table A and table B, and then selects columns just from A (or does `SELECT DISTINCT a.*`). The only reason to do `DISTINCT` after the join is to remove duplicates introduced by a one-to-many join. If we know that the join did not multiply rows – for instance, if B had at most one matching row per A (perhaps B's join column is a primary key or unique) – then the DISTINCT is redundant. Or if B's presence is only to filter A (like a semi-join), we could rewrite the query to use an `EXISTS` subquery or a semi-join which inherently doesn't duplicate A's rows. **Why the DB won't:** If the optimizer isn't aware of B's uniqueness, it must assume the join could duplicate rows, so it can't remove the DISTINCT safely. An LLM, however, can infer or be told that B's join key is unique (or that this join is effectively a filter). It could then rewrite `SELECT DISTINCT a.* FROM A JOIN B ON ...` to `SELECT a.* FROM A WHERE EXISTS (SELECT 1 FROM B WHERE ...)` or use a `SEMI JOIN` if supported. This avoids generating a potentially large intermediate result

only to de-duplicate it [7]. Even if uniqueness isn't declared, the LLM might guess from naming or usage (e.g. joining on something that looks like an ID). It can present this as a suggestion with a confidence level, or ask for confirmation of uniqueness. The result is a query that is simpler and often faster (especially if B is just a filtering table). *Detection cues:* a `DISTINCT` on a query that involves a join, especially where only one table's columns are ultimately needed. That hints the DISTINCT might be compensating for duplicate join combinations.

- **Removing Redundant Joins via Functional Dependency:** In large SQL queries, sometimes a table is joined even though all the information it provides is already available or not actually needed. For example, joining a dimension table just to fetch a foreign key that you already have, or joining to get a column that is functionally dependent on an existing column. If table C is joined but only columns from C's primary key (which is already present in the join from the other table) are used, that join doesn't change the result – it's redundant, serving perhaps as a sanity check or leftover from an ORM-generated query. A DB optimizer typically cannot assume a join is redundant unless there is a declared foreign key and perhaps a NOT NULL constraint (to ensure it doesn't filter out any rows). LLM can again use schema metadata or even simple reasoning: "They join Customer table just to get customer_id which they already had – likely the join is unnecessary." It can propose dropping that join or at least dropping the columns from that join to allow the optimizer to remove it. Another case is if after a join, the query only selects columns from one side and the join condition implies at most one match, then the other side is not needed (this is join elimination). *Detection cues:* patterns where a table is joined but none of its non-key columns are used, or the output of the query doesn't actually include anything from that table. The LLM would, of course, verify via metadata if possible (like confirm that the join is a foreign key relationship that doesn't filter rows). This rewrite can **reduce I/O and computation** by cutting out whole tables from the query plan [8] (the LITHE example in the introduction showed a redundant join on a PK-FK that optimizers didn't remove [8]).

- **Simplifying GROUP BY with Key Constraints:** Another place constraints matter is grouping. Developers sometimes group by more columns than necessary. For example, suppose you group by (user_id, user_name) to get counts per user, but *if user_id is a primary key for users, then user_name is functionally determined by user_id.* Grouping by both is redundant – grouping by just user_id would produce one row per user, and you can still select user_name (since it's the same for that user_id). If the DB doesn't know about that functional dependency (no declared key or unique index on user_id perhaps), it won't remove user_name from the grouping list. An LLM can infer that user_id is a key and user_name is an attribute of that key, and rewrite the query to group only by user_id (or whatever minimal key). This can substantially reduce the grouping workload (fewer columns to hash or sort on) and even allow better indexing usage. Another example: grouping by a date and an hour field, when the hour is derived from the date or the combination is unique anyway – you could group just by the full timestamp if that uniquely identifies the hour bucket. *Detection cues:* multiple columns in GROUP BY that look like they come from the same table where one is an obvious primary key or unique identifier. The LLM would likely need schema info (primary keys or unique constraints) to be confident in suggesting this rewrite. **Why DB won't:** Without declared functional dependencies, the engine can't assume it's safe to drop group-by columns – it could change results if the assumption is wrong.

- **Assuming Data Constraints for Optimizations:** More generally, an LLM optimizer can leverage assumptions like *"this column is unique"* or *"these two tables always match one-to-one"* to enable rewrites. For instance, if it knows (or asks and the user confirms) that a certain ID field is unique, it can safely eliminate duplicate-elimination steps or unnecessary aggregations. If it knows a foreign key relationship, it can remove a join that only served to restrict to matching keys (since the existence is guaranteed by the data integrity). It can even do things like **pre-aggregate** data

under the assumption that certain keys cover all cases (though that crosses into the next category of cross-statement reasoning).

LLM Advantage: The LLM can use **out-of-band knowledge** (like schema metadata, or even statistical analysis of the data) to perform optimizations a normal optimizer can't. It essentially can say, "If we assume X is true about the data, we can do Y to simplify the query." This yields big payoffs – eliminating whole joins or needless DISTINCT/GROUP BY operations can drastically speed up queries. The key is that the LLM can be made aware of constraints either through a metadata API or by asking the user, and then act on them. Traditional optimizers are blind if the constraints aren't formally declared, whereas an LLM agent can be much more flexible in using plausible but undeclared information.

4. Cross-Query and Cross-CTE Reasoning

Database optimizers mostly operate within the boundary of a **single SQL statement**. They don't easily combine or rearrange logic across multiple statements, Common Table Expressions (CTEs), or subqueries if it means breaking those boundaries. In some engines, a CTE is treated as a *optimization fence* – meaning the optimizer will execute the CTE as given and not pull up predicates or merge it with the outer query [9]. This is done for logical clarity or to preserve semantics, but it can prevent otherwise possible optimizations. An LLM-driven approach, not bound by the engine's strict phases, can reason across these boundaries. For example:

- **CTE Inline vs. Materialization:** Many SQL queries use **WITH (CTE)** subqueries for readability. Some databases (e.g. PostgreSQL before v12, or as a rule for debugging) will *always* materialize a CTE – computing it fully and storing results – even if it's only used once or could be merged into the main query [10]. This can be horribly inefficient if the CTE is large and the outer query filters it further. Other engines like SQL Server might inline by default, but could still materialize if the CTE is referenced multiple times or if it's non-deterministic. An LLM optimizer can detect when a CTE is acting as an **unnecessary barrier**. If a CTE is used only once, it can suggest inlining it (i.e. replacing the CTE reference with the CTE query) so that filters can push down. Conversely, if a CTE *is* used multiple times (and the engine isn't smart enough to reuse the computation or if it recomputes it for each usage in some engines), the LLM might suggest explicitly materializing it (e.g. creating a temporary table or using a `/*+ materialize */` hint, depending on engine) to avoid duplicated work. The ability to choose **inline vs materialize** depending on context is something a human tuner often does, and an LLM can mimic that. Your system already tracks some patterns like this (e.g. SQL-CTE-001/006 for materialization fences). The LLM can extend it by analyzing the size of the CTE, presence of filters on it later, and the database's behavior. *Detection cues:* a CTE that is only referenced once and followed by a filtering WHERE clause or join in the outer query – likely better to merge it. Or a heavy CTE used in multiple parts of the query – possibly better to materialize once.

- **Eliminating Redundant Recomputations Across Subqueries:** Optimizers generally optimize each part of a query, but might not catch that two different subqueries (or two parts of a UNION, or two different CTEs) are doing the same thing. For instance, if you have a complex filter condition repeated in multiple subqueries, the engine might scan the same large table twice, once for each subquery. Common subexpression elimination across the entire query is not guaranteed, especially if the subqueries are in different parts of a CTE or a UNION block. An LLM can identify that pattern and refactor the SQL to **compute the common part once**. For example, if two subqueries both select from a big table `Sales` with `WHERE date >= '2023-01-01'` (the same filter) but then do different aggregations, an LLM could introduce one CTE that selects the filtered subset of `Sales` and then have the two sub-operations use that CTE. This way, the

expensive scan and filter is done once. A human optimizer often does this by hand when they see repeated logic. The LLM can also conversely *unroll* things if needed: sometimes breaking a single query into pieces can avoid repeated scanning due to how the engine executes it. For example, it might recommend writing results of an expensive subquery to a temporary table and reusing it, rather than letting the optimizer evaluate it repeatedly. These decisions are tricky because not all engines handle CTEs or temp tables the same way, but with engine-specific knowledge (see next section) the LLM can navigate that. *Detection cues:* multiple subqueries or SELECTs in a query text that have identical `FROM ... WHERE ...` clauses on a large table, or identical expressions. The LLM could suggest refactoring to a single CTE or using a UNION of precomputed results to avoid double work.

- **Cross-Statement Optimization (Workload-Level):** While traditional optimization stops at the query boundary, an LLM optimizer could even look at **sequences of queries or the overall workload**. For example, if query A always runs right after query B and B's result is used in A, maybe they can be combined or B's result cached for A. Or if a report is generated by running 5 similar queries, maybe a single combined query or a precomputation could serve all. This is more of a workload scheduling idea and often not in scope of a single query optimizer, but an LLM has the flexibility to make such holistic suggestions (this overlaps with section 6 on workload awareness). For now, the main "cross-boundary" focus is on **CTEs and subqueries within one query**.

Why DB Optimizers Won't: Many engines treat each CTE as an independent unit for clarity or to avoid unpredictable performance. They typically won't pull a filter from outside into the CTE (some newer versions of databases do try to optimize this, but it's not guaranteed). They also usually won't do global common subexpression elimination across different parts of a query if it's not obvious, due to complexity. The optimizer's search space is exponential already; considering merging separate subqueries might be beyond its scope unless explicitly written in the SQL.

LLM Advantage: The LLM, with its more holistic view, can **reorganize the query** in ways the engine won't. It can decide to inline a CTE to enable other optimizations (removing the "fence"), or conversely isolate a repeated sub-logic to compute once. It can weigh the **engine-specific behavior** (like "Engine X materializes CTEs, so better inline them if possible" or "Engine Y can cache CTE if used multiple times, so use that"). This kind of cross-query reasoning is akin to what expert tuners do manually by rewriting queries. The LLM can apply those techniques automatically. In essence, it can achieve a form of **semantic common subexpression optimization** beyond what the database planner might do, by rewriting the SQL itself. This can eliminate redundant work and significantly speed up queries that have repeated patterns.

5. Engine-Specific Planner Blind Spots

Each database engine has its own quirks – certain query patterns that confuse the optimizer or trigger a suboptimal plan. These are not logical issues, but rather limitations or heuristics in the planner's implementation. Human experts often learn these ("lore" about a particular DBMS) and adjust their SQL to avoid known pitfalls. An LLM-driven system can be **taught these engine-specific patterns** and rewrite queries to steer the planner toward better decisions or work around the quirk. Some examples:

- **Preventing Window Functions from Blocking Pushdown:** In some engines (e.g. DuckDB, older versions of others), using a window function can inadvertently block index or partition **predicate pushdown**. For instance, if you have `SELECT ... FROM LargeTable WHERE part_key = '2023-01' AND ...` and you add a `ROW_NUMBER() OVER (PARTITION BY ...)` in the

select list, the engine might decide it must scan the whole table and only apply the `WHERE` filter *after* computing the window (because the window might need the full partition) [11]. This is a planner limitation – ideally it should apply the filter first since it's on the partition key, but some systems don't. The LLM can recognize this pattern (window function present + simple filter on same partition key) and rewrite the query to enforce the filter first. One way is to isolate the filtered subset as a subquery/CTE **before** applying the window. For example: `WITH subset AS (SELECT * FROM LargeTable WHERE part_key = '2023-01') SELECT ..., ROW_NUMBER() OVER (...) FROM subset ...`. If the engine treats CTEs as fences, the LLM might instead suggest doing the filter in a derived table or an inner subquery so that it definitely happens before the window. By rewriting the SQL, we ensure the window only sees the already filtered data, working around the pushdown issue. This pattern (which you noted as "SQL-DUCK-012") is a prime example of encoding a *workaround* for a planner quirk. *Detection cues:* presence of window functions (especially partitioned by a key that's also filtered in WHERE), and known engine hint that pushdown is not happening. The LLM can be pre-loaded with knowledge like "DuckDB cannot push filters past a window operator [11], so rewrite to filter first."

- **Join Strategy Hints via Query Reshaping:** Many databases have multiple join algorithms (hash join, nested loop, merge join) and strategies (which table to build or probe). The optimizer's choice depends on statistics and estimates. Sometimes the optimizer picks a suboptimal plan (e.g. building a huge hash table on a large table and probing with a small one, when the opposite would be better). If the engine doesn't support explicit hints or if hints are not desirable, we can **rewrite the query** to influence the planner's decision. For example, if we suspect a join order is wrong, the LLM could rewrite the FROM clause order or nest the joins in a way that forces a particular association (some optimizers take the written join order as a hint in absence of better info). Or the LLM might introduce a deliberate **aggregation or pruning** before a join to reduce the size of a join input, coaxing the optimizer to switch strategy. Another trick: turning an OR condition into a UNION ALL of two queries can sometimes let the optimizer use indexes separately for each branch, whereas an OR prevented index usage. This *query shape change* effectively gives the optimizer a different set of options. For instance, `WHERE col=X OR col=Y` could be rewritten as `SELECT ... WHERE col=X UNION ALL SELECT ... WHERE col=Y` (assuming it's safe and duplicates aren't an issue) – this can be faster if each single-filter query hits an index (the engine might otherwise do a full scan for the OR). **Why DB won't:** The optimizer usually doesn't restructure ORs into UNION ALL on its own, and it won't pre-aggregate or pre-filter data unless it's logically the same query. An LLM can apply these human-like tricks: "divide this query so that each part is simpler for the planner." It essentially acts as a *planner hint* but implemented as SQL rewrite. *Detection cues:* evidence from an explain plan that a join order is bad (e.g. a huge intermediate result or a wrong cardinality estimate). The LLM could have rules like "if a join has a highly selective filter on one table but it's not being applied early, rewrite the query to force that table to be joined later or filtered first." Also, OR conditions across different columns or tables that cause a scan – rewrite as UNION as mentioned.

- **Improving Sargability with Computed Columns or Casts:** Sometimes queries are slow because they are not **sargable** – i.e., they can't use indexes due to the way the WHERE clause is written. Classic cases: wrapping a column in a function (`WHERE DATE(timestamp) = '2023-01-01'`) or an implicit type cast (`WHERE numeric_col = '123'` where '123' is a string literal) can prevent index usage. The DB optimizer can't fix the SQL; it will just do a sequential scan if it can't use the index. An LLM optimizer can suggest a rewrite like "instead of `DATE(ts) = '2023-01-01'`, write `ts >= '2023-01-01' AND ts < '2023-01-02'`" so that the filter is on the raw column and can use an index or partition pruning. Or it might suggest creating a **computed column** (persisted) for `DATE(ts)` if this query runs frequently,

then index that column – but that bleeds into schema changes (see section 8). Another example: if queries often do `LOWER(email) = 'abc'`, the LLM could recommend storing a lowercased email column or index on lower(email). These are outside the scope of what the optimizer can do by itself (it can't create new indexes or columns on the fly!). Similarly, if a query joins on an expression (e.g. join on `LEFT(date,7)` to match year-month), the optimizer can't use an index on the raw date; an LLM could propose a redesign (store year-month as a column). In summary, the LLM can identify *non-sargable patterns* and rewrite them into sargable ones. *Detection cues:* Functions or casts around columns in WHERE clauses (`WHERE UPPER(name) = ...`, `WHERE CAST(col AS ...) = ...` etc.), or pattern like `WHERE col1 OR col2 = value` (which could be rephrased). The LLM sees these and knows they block index usage. It can rewrite accordingly or suggest adding an index on the expression if rewriting is not straightforward.

- **Other Engine Quirks:** There are countless specific quirks – e.g., maybe "MySQL can't optimize `COUNT(DISTINCT) with multiple columns` well" or "Postgres planner sometimes doesn't switch to an index for `IN (...)` if too many literals". An LLM can contain a knowledge base of such things and apply known fixes (like "for MySQL, if large IN list, consider splitting or using EXISTS", etc.). Another example: **Cartesian join prevention** – if a query accidentally misses a join condition, some engines won't notice and will do a huge Cartesian product. The LLM can warn and suggest adding the missing join predicate (this overlaps with correctness fixes in section 7, but it's engine-agnostic too). The key is customizing the rewrite to the *specific database's behavior*. For instance, "this particular pattern blocks pushdown in DuckDB, rewrite it like this" or "Snowflake doesn't propagate certain filters across JOIN, so do it in a subquery instead."

LLM Advantage: Essentially, the LLM can be imbued with **tribal knowledge of query tuning** for each DBMS. It acts like a seasoned DBA who knows "if you do X in this engine, the planner will choke, better write Y instead." By encoding these heuristics, the LLM can **rewrite the query to avoid known planner pitfalls**, resulting in better execution plans without needing explicit optimizer hints. This is powerful because it can encapsulate best practices and workaround patterns that are not obvious to regular users. Over time, as engines improve, these patterns might change, but the LLM's knowledge can be updated more easily than waiting for users to learn every nuance. It's like having a constantly updated manual of "do's and don'ts" for the SQL dialect at hand, and the LLM applies it on the fly.

6. Workload-Aware Refactoring

A database optimizer typically looks at a single query in isolation and tries to make it run fast. It doesn't consider how often the query runs, what context it's used in (ad-hoc analysis vs live dashboard vs ETL batch), or opportunities to restructure the workload. An LLM-driven approach can take into account **the broader usage patterns** and propose more radical changes that improve performance at a system or workload level. These go beyond tweaking a single SQL and often involve changes to how data is organized or retrieved across queries. Some high-level ideas:

- **Pre-Aggregation and Summary Tables:** If the LLM notices that a query (or set of queries) is repeatedly scanning a large fact table to compute metrics at a higher grain (say daily summaries, or totals by category), it might suggest creating a **materialized aggregate table** that stores these results, updated on a schedule. For example, if every day 1000 dashboard calls are summing last month's sales by region, the LLM could propose a new table that already has `sales_by_region_by_day` and just query that. Traditional optimizers won't suggest this because it's a design change, not a per-query fix. The LLM can identify the pattern ("they're always grouping by day and region on the main table") and reason that maintaining a summary table or materialized view could cut the workload. It can either rewrite queries to use an existing

summary (if the system has one and the LLM has schema info), or output a recommendation to the user like "Consider pre-aggregating this data into a summary table; the LLM can then automatically use that table for the dashboard queries." Modern cloud warehouses often encourage this pattern (like creating *aggregations for BI*), but automating it is a next step. *Detection cues:* frequent queries with the same GROUP BY on a large table, especially if they scan large date ranges repeatedly.

- **Caching and Reusing Results:** If a query runs very often (e.g. dozens of times per minute with slightly different parameters), the LLM might suggest using parameterized queries or caching. For instance, if only a date or an ID parameter changes, perhaps the application could call a stored procedure that caches recent results or the DB could benefit from prepared statements. An LLM can rewrite literal values into parameters to encourage plan reuse (some engines treat two queries with different literal values as separate plans, which defeats caching). It could also point out that "This query runs 10k times a day; consider using a caching layer or result cache." Some databases have result caching that returns cached results if the underlying data hasn't changed – the LLM could suggest enabling that or structuring the query in a cache-friendly way (e.g. avoid non-deterministic functions that would disable caching). These are optimizations at the **application level** often, but the LLM optimizer can bridge that gap by making the developer aware. *Detection cues:* multiple similar queries in the workload (the system would need to provide the LLM with that info) or a single query text with varying constants.

- **Breaking Down a Monolithic Query:** Sometimes a single giant SQL query is doing too much in one go – maybe it joins many tables and applies complex logic all together. While a single SQL is often ideal for set-based processing, there are times when **splitting it into stages** can be beneficial. For example, if one part of the query significantly reduces the data (a highly selective filter or an aggregate), it might be better to materialize that intermediate result and then join or process further, rather than letting the optimizer carry all the joins and filters together. Traditional optimizers *do* pipeline operations and push predicates as much as possible, but they won't decide to "stop early and store intermediate" because that's not part of the declarative query – except in some cases of automatic temp spool for repeated subplans. An LLM can explicitly suggest rewriting: "First, filter this big table and store the result in a temp table, then join with the other tables." This can help if the optimizer was repeatedly scanning the big table for multiple parts of the query or if the join order is hard to get right. Another scenario is simplifying logic: maybe a query has a complicated CASE expression or subquery that could be precomputed, etc. By staging the query, you sometimes give the optimizer simpler pieces to chew on and you can create indexes on intermediate results. Of course, this increases I/O because you write out and read, but if it avoids a disastrous plan or enables parallelism, it might be worth it. The LLM can weigh these trade-offs or even run some what-if tests (if integrated with the DB runtime). *Detection cues:* extremely large and complex SQL with many joins and subqueries, or the presence of the same large sub-expression multiple times. The LLM might propose a step-by-step approach – which essentially is the LLM acting as a *human query planner*, deciding to break the problem into subqueries manually.

- **Workload-Level Prioritization:** The LLM could also notice if certain queries are more latency-sensitive (say queries that drive user-facing dashboards) and suggest optimizations geared for low latency, versus others that are batch jobs where throughput (total work done) might matter more than single-query latency. For instance, a frequently run query might benefit from an index even if it makes writes slower, whereas a one-off analytical query might not justify an index. The LLM can incorporate these considerations if given metadata like `query_frequency` or a hint of interactive vs batch. It might say "This query runs very often; it might be worth adding an index

on column X to speed it up," whereas the DB optimizer doesn't consider future queries at all. This crosses into physical design suggestions (section 8).

LLM Advantage: The LLM is not constrained to a single query view – it can **learn from the workload**. It operates more like a database engineer thinking about caching, precomputation, and query patterns. By doing so, it can achieve improvements that a per-query optimizer cannot – sometimes improving overall system performance by orders of magnitude through caching or pre-aggregation strategies. This effectively turns the optimizer into a full **tuning assistant** that considers when to use different design patterns. While implementing such suggestions may require user intervention (e.g. creating a summary table or adding an application cache), the LLM can integrate with automation (like automatically creating materialized views if allowed). This bridges the gap between query optimization and physical design optimization.

7. Correctness-Risk Anti-Patterns and Fixes

Not all query issues are about performance alone – some are about correctness pitfalls that also impact performance or reliability. Traditional optimizers usually don't warn or fix these; they assume the query is what it is. An LLM-driven system, however, can identify patterns that are *likely mistakes or dangerous choices* and suggest safer alternatives that preserve correctness and often improve performance as well. These are less about making things faster by transformation, and more about **preventing disastrous slow or wrong behaviors**. They are valuable because an LLM can explain the issue to the user (education) and offer a fix. Some common ones:

- `NOT IN` **with NULL Pitfall:** A known SQL trap is using `NOT IN` subqueries when the subquery might produce a NULL. For example, `SELECT ... WHERE id NOT IN (SELECT fk FROM other_table ...)`. If `other_table.fk` contains any NULL, the `NOT IN` will return no rows at all (because the comparison "X NOT IN (...NULL...)" yields UNKNOWN for all X). This is often *not what the user intended* – it silently fails to return results. Additionally, even if NULLs aren't present, some engines handle `NOT IN (subquery)` by converting to a form that can be less efficient than a semi-join. The safer pattern is to use `NOT EXISTS` with a correlated subquery, or an anti-join (`LEFT JOIN ... IS NULL`). The LLM can spot `NOT IN (SELECT ...)` and warn that if the subquery might have NULL, the results will be incorrect [7]. It can then rewrite the query to use `NOT EXISTS` (which ignores NULLs in the subquery by design) or add a condition in the subquery `AND value IS NOT NULL` to explicitly guard against it. As a bonus, the rewrite to `NOT EXISTS` often has the same or better performance, and avoids this logical bug. This pattern is something you already track (SQL-WHERE-005/011 perhaps). The LLM can handle it by simply explaining and providing the corrected query.

- **Overly Broad** `OR` **Conditions:** Using `OR` in WHERE clauses can sometimes **prevent index usage** and cause the optimizer to do a full scan or a heavy union all behind the scenes (especially if the OR spans different columns or is highly selective in one part but not another). Aside from performance, a huge OR condition (like dozens of ORed predicates) can lead to extremely complex plans or even exponential explosion in some planners (each OR might be a separate branch). The LLM can detect a large OR list and suggest alternatives: one common fix is to rewrite `OR` conditions as a UNION of separate queries (as discussed in section 5) if that allows each part to use indexes or simplifies the logic. Or, if appropriate, transform it into an IN list or a between, etc. Another scenario: conditions like `(condition1) OR (condition2)` can sometimes be refactored by *distribution* or other logical means to allow pushdowns. The LLM can carefully do this while ensuring equivalence. This is both a performance improvement and a safeguard – complex OR conditions can lead to *plan explosion*, meaning the optimizer

enumerates many possibilities and spends too long, or picks a very bad plan. The rewrite can simplify the planning. The LLM would also caution that splitting by UNION ALL means duplicates have to be considered (if the original OR conditions aren't mutually exclusive) – it might add a DISTINCT or otherwise, or if it's known that the OR branches are disjoint, then it's fine. This is advanced, but definitely an area an LLM can help where static optimizers often leave as-is (except trivial OR to IN-list normalizations).

- **Implicit Type Casts and Non-Sargable Functions:** We touched on this in engine-specific, but it's worth listing as a general correctness/performance issue. If a query compares mismatched types (e.g. string literal to number column), the DB might implicitly cast one side. Depending on the engine, this could result in a full scan (because e.g. casting the column value for every row to compare). It can also produce incorrect comparisons if locale or precision is involved. The LLM can flag these and advise to explicitly cast the literal instead of the column, or better yet, supply the literal in the correct type (e.g. `... WHERE numeric_col = 123` instead of `... = '123'`). Similarly, if a query applies a function to a column in the WHERE clause (e.g. `WHERE DATE(ts) = ...` or `WHERE LOWER(name) = 'abc'`), it's both a performance issue and a potential pitfall if one expects an index to be used. The LLM's rewrite: use a range condition for dates, or use a computed lower-case column or at least note that an index on `LOWER(name)` would be needed. This improves performance and avoids surprise where the query is slow because the index on `name` wasn't used due to the function call.

- **Missing Join Conditions (Cartesian Join):** A classic bug is forgetting a join condition, leading to a Cartesian product of two tables. This can blow up result size exponentially and is almost never intended (unless specifically doing a CROSS JOIN for a reason). The LLM can easily notice if a `JOIN` lacks an `ON` clause (or using old-school comma join without a where condition linking tables). It will alert "It looks like TableA and TableB are being joined without a predicate, which will produce a Cartesian product" and suggest the likely missing condition if it can guess (maybe by matching column names, or using foreign key info). Even if it can't guess, it will at least flag it so the user can fix it. Some engines might execute a Cartesian join without warning, causing huge slowdown or out-of-memory. So this is a **correctness and performance** fix in one. No traditional optimizer will add a join condition that isn't there; at best the user gets a result with all combinations or a timeout. The LLM can act as a safety net here.

- **Unbounded Result Sets (lack of LIMIT or filter in huge queries):** If a query unintentionally could return an extremely large number of rows (e.g. missing a WHERE on a table scan when it was probably needed), the LLM might warn. This is more speculative – the optimizer doesn't care, it will try to execute whatever. But if in context the user likely meant to restrict it (for example, a query pulling user logs without a user ID filter might be a mistake), the LLM could question it. This veers into intent analysis; perhaps better as a caution: "Do you really want to select 10 million rows? Consider adding a WHERE clause or LIMIT for safety." It's not an equivalent rewrite, more of a recommendation.

- **Partition Pruning Issues (Timezone/Date Truncation traps):** If the database uses partitioning (e.g. by date), queries need to be written in a way that the partition pruner can understand. A common trap is applying a transformation to the partition key, which can prevent partition pruning. For instance, if a table is partitioned by a date column `event_date`, and the query does `WHERE DATE(event_timestamp) = '2023-01-01'`, if `event_timestamp` is a `TIMESTAMP` partitioned by day, applying `DATE()` might disable pruning (the optimizer sees a function, not a direct partition key comparison). The result: the query will scan all partitions and then filter, which is slow. The correct approach is

`WHERE event_timestamp >= '2023-01-01' AND event_timestamp < '2023-01-02'`. The LLM can detect this pattern (function on a partition key in the predicate) and rewrite to an equivalent range condition that allows partition pruning. Similarly, using the wrong timezone or not aligning with partition boundaries can cause subtle bugs or performance hits. The LLM might also mention if using `TIMESTAMP WITH TIME ZONE`, you have to be careful with date truncation in SQL. These are esoteric for the average user but exactly the kind of "gotcha" an expert system can catch.

LLM Advantage: Many of these are not about making the plan faster through a clever transformation, but about **catching errors or inefficiencies that the DB optimizer silently ignores**. The LLM can effectively serve as a **SQL review assistant**, pointing out risky patterns (like `NOT IN` with NULL) and offering alternatives (`NOT EXISTS` which is both safe and often faster [7]). This not only improves performance (in cases like the OR -> UNION or making queries sargable), but also ensures correctness and robustness. Importantly, the LLM can *explain why* – something optimizers don't do. For example, it can explain "Your NOT IN subquery might return no results because of a NULL; rewriting to NOT EXISTS avoids that null issue [7]." This educational aspect helps developers write better SQL in the future. In a sense, this turns the optimizer into a teacher that prevents "slow and wrong" queries proactively.

8. High-Impact LLM-Only Optimization Opportunities

Beyond the specific patterns listed, some broad areas stand out as **high leverage improvements** that an LLM-driven approach can provide. These are the next frontiers to focus on, because they often yield substantial real-world gains beyond what is achievable with conventional methods (and beyond the patterns already implemented in your system):

- **Constraint-Driven De-Duplication Removal:** As discussed in section 3, leveraging primary key/foreign key or unique constraints to eliminate unnecessary `DISTINCT` operations, redundant GROUP BY columns, or even whole joins can be extremely impactful. Many SQL queries generated by tools or ORMs include needless duplicates removal or grouping just to safeguard correctness, when the data inherently prevents duplicates. By confidently identifying where a result is naturally unique (e.g. a join of fact to dimension on key will not duplicate the fact if it's 1-to-1, etc.), the LLM can remove `DISTINCT` and extra grouping. This tends to **speed up queries massively** by avoiding sorting or hashing for duplicate elimination. It's "LLM-only" in the sense that the optimizer won't do it without declared constraints, but an LLM can infer it with high probability from context. Implementing a robust way for the LLM to know or ask about uniqueness will unlock this class of optimization widely.

- **Workload-Level Refactoring and Caching:** Patterns that improve not just one query but the entire workload are golden. For example, if the LLM recognizes a set of queries all filter by `customer_id` and aggregate by day, it can suggest creating a per-customer daily summary table that all those queries can draw from, rather than each query scanning the entire data. Or if a nightly ETL and a daily report could share some computations, suggest restructuring to reuse intermediate results. These changes might involve materialized views, caching query results, or splitting heavy tasks into incremental updates. While these go beyond a single query rewrite, the LLM can output recommendations or even pipeline instructions to implement them. This is something a human architect would do by analyzing logs and usage; having the LLM do it accelerates that process. The payoff is huge in terms of saved compute and faster user experience.

- **Latest/Top-N per Group Intent Patterns:** We already covered them in section 1, but it's worth emphasizing: queries asking for "the latest record" or "top N records per category" are extremely common in analytics and application logic, and they are often written in inefficient ways (self-joins, subqueries) that scale poorly. By adding robust handling for these – recognizing all the various ways people write them and rewriting to the optimal window function or lateral join approach – the LLM optimizer can improve a *lot* of queries out of the box. It's a high-impact pattern family because these queries often hit large tables and can go from minutes to seconds with the right rewrite. Ensuring the LLM handles tie-breaking (e.g. if two rows have the same timestamp, maybe take the highest ID as "latest") is important for correctness. This family is partly implemented (you mentioned you have some rules like SQL-DUCK-014 for grouped top-N to LATERAL), and extending it further will cover more use cases.

- **Engine-Specific Magic "Recipes":** Focusing on each target engine's known blind spots (section 5) and coding those into LLM prompts can yield quick wins. For example, ensure the LLM knows the dozen most common Postgres planning edge cases, the common MySQL/MariaDB ones, etc. We already listed some (CTE fence, window pushdown, OR-to-UNION). Another might be *forcing index usage by hinting via a* `INNER LOOP JOIN` *rewrite* (like sometimes selecting in a subquery to force nested loop). These are not general, but each one, when applicable, can turn a 60-second query into a 1-second one. "LLM lore" is an asset – capturing it means the optimizer's power grows with collective tuning knowledge.

- **Schema Change Suggestions (Beyond Query Text):** An advanced but very useful output the LLM can provide is advising changes to the **schema or physical design** to improve performance. For example: *computed columns*, indexes, partitioning, or clustering keys. The LLM can say "Query X would benefit greatly from an index on (customer_id, order_date) because it does lookups by customer and date range." Or "All these queries filter on LOWER(product_name); consider adding a generated column for lowercase name and index it, or always store names in one case." Another: "Your table isn't partitioned by date but you always query one month at a time – consider partitioning, it will help prune data." Traditional optimizers won't say any of this – they just suffer in silence with non-ideal schema. Some modern systems have advisors, but an LLM can integrate it into the conversation about rewriting: after rewriting the SQL, it can append "and if possible, adding index on X would help further." Since the question is about aiding the LLM optimizer itself, these suggestions could be a separate channel (like not automatically applied, but presented to a developer or DBA). However, including this capability closes the loop from just query rewriting to holistic optimization. It empowers the LLM to not only rewrite queries for current schema, but also hint at how changing the environment could allow even better rewrites or performance (like "if there was a unique constraint on column Y, I could eliminate that DISTINCT safely – maybe add the constraint or ensure it in data").

In summary, focusing on the above high-impact families (unique-driven simplifications, workload-aware optimizations, intent-based top-N patterns, engine-specific fixes, and schema/index suggestions) will give the **biggest real-world gains** in an LLM-driven optimizer. These are areas where humans typically make a big difference in tuning, so encoding them into your system will let it approach expert-level performance tuning.

9. Empowering the LLM Optimizer with Metadata and Context

To safely and effectively implement the rewrites discussed, an LLM-based optimizer benefits enormously from having additional **metadata and context** about the database and usage. Unlike a built-in optimizer which has direct access to statistics and constraints, the LLM operates at the query-

text level unless we enrich its input. You should consider providing a **"metadata contract"** – a structured set of information that your pipeline feeds into the LLM prompt alongside the query. Key pieces of metadata include:

- **Keys and Uniqueness Constraints:** Knowledge of which columns are primary keys, unique keys, and foreign keys is vital. This allows the LLM to confidently apply many rewrites (e.g. remove DISTINCT because it knows a join key is unique, or use an EXISTS because it knows a one-to-many won't duplicate). Even if not all constraints are declared, you might infer likely keys from data profiling (or let the user hint them). Feeding these in a standardized way (like a list of tables with their PK, list of FKs, etc.) will let the LLM reason about uniqueness and referential integrity in its rewrites.

- **Ordering and Monotonicity Info:** If possible, indicate which columns can define a stable ordering for pagination or "latest" queries. For example, mark a column as a timestamp that increases, or an ID that is sequential. Also whether that column is unique (or combination of columns). This helps the LLM safely apply the keyset pagination or latest-N rewrites. E.g. *"Table Orders: primary key (order_id). order_date is chronological, but not unique without order_id."* Then the LLM knows to use (order_date, order_id) as the ordering key for stable results.

- **Approximation Tolerance & Use-Case Hints:** Provide a flag or threshold for whether approximate results are allowed. For instance, a setting `approx_ok = true` (perhaps with error tolerance like ±5%) if the query comes from a dashboard, versus `approx_ok = false` for, say, financial reports. This guides the LLM on whether it should even consider those semantic-changing rewrites in section 2. Similarly, a hint if fresh/stale data is acceptable (for recommending materialized views or caches). A query running in a daily batch could maybe use yesterday's precomputed results if needed, whereas an interactive ad-hoc query expects real-time data.

- **Workload Frequency and Importance:** If the system can tag how frequent or important a query is (perhaps via monitoring data), include that. For example, *"Query Q runs 10k times per day and is user-facing (latency critical)"* vs *"Query Z runs once a day as a background report (throughput important, latency less so)"*. The LLM can use this to prioritize latency optimizations (like index usage, avoiding heavy compute) for the high-frequency, latency-critical queries, and it might choose plans that are optimal for throughput (like a scan that is fine to take longer as long as it can handle volume) for batch queries. This can influence whether it suggests caching or splitting etc. In essence, it contextualizes the rewrite goals (fast response vs minimal resource usage vs one-off).

- **Engine Profile and Limitations:** Clearly identify the target database engine (PostgreSQL, MySQL, DuckDB, Snowflake, etc.) and any known relevant behaviors. You might encode a list like *"Engine: DuckDB. Quirks: CTEs materialize by default; window functions block partition pruning; supports vectorized execution but no indexes; etc."* or *"Engine: PostgreSQL 15. Quirks: Most CTEs are inlined now, but watch out for lateral joins affecting estimates; supports indexes on expressions; has parallel query execution,"* and so on. The LLM can then tailor its suggestions – e.g., it won't suggest something unsupported (like `QUALIFY` clause on an engine that doesn't have it, or index advice on DuckDB which doesn't use indexes in the traditional way). It will also know which patterns from our list are applicable or not. This **engine context** is crucial for making the rewrite both valid and effective.

- **Statistics (optional):** If available, high-level stats like table row counts, or predicate selectivity info can help. For example, knowing that a filter `status = 'inactive'` keeps only 0.1% of rows means the LLM might pre-filter that first. Some research (like the LITHE system [12]) used actual selectivity to pick rewrites (Exists vs In). If you can supply approximate selectivities of subquery predicates, the LLM can choose e.g. `EXISTS` vs `IN` as per rule R5 [13]. This is an advanced input, but even coarse info (like "this table is 100x bigger than that table") can guide join order choices.

By establishing this metadata input, your LLM optimizer can say, for example: *"I propose this rewrite with confidence 0.92 because (customer_id, order_ts, order_id) is a unique key and approx_ok=false, so I'm keeping it exact."* Essentially, the LLM can justify and tailor its transformations with awareness of the database's reality, rather than guessing. This makes its suggestions much safer and more targeted, addressing the main concern with LLM rewrites (ensuring semantic equivalence or acceptable differences).

Conclusion

Traditional query optimizers remain a critical component for executing queries efficiently under the hood, but they have clear boundaries defined by rigid rules and lack of context. LLM-driven optimizers have the exciting ability to **transcend those boundaries** – understanding the intent behind a query, using external knowledge of the data and workload, and employing flexible reasoning to rewrite SQL in ways that human experts would. The combination of rule-based and LLM-based optimization can yield the best of both worlds: *sound, local optimizations* handled by the engine and **creative, holistic improvements** proposed by the LLM. Real-world studies already show substantial performance gains using LLM rewrites as an advisory layer [3], and systems like R-Bot demonstrate these techniques in enterprise settings with multi-fold latency reductions [4]. By expanding the repertoire of patterns (especially the high-impact ones outlined) and feeding the LLM with the right metadata, we can significantly improve our optimizer. The ultimate goal is a self-improving system where the LLM continuously learns new optimizations and applies them safely – effectively **bridging the gap between the user's intent and the database's execution**. This allows queries to run faster and more reliably, not by magic, but by encoding the very expertise and patterns that seasoned engineers have known for years, now scaled up through AI.

---

[1] [2] [3] [8] [12] [13] [2502.12918] Query Rewriting via LLMs
https://ar5iv.labs.arxiv.org/html/2502.12918v4

[4] vldb.org
https://www.vldb.org/pvldb/vol18/p5031-li.pdf

[5] How We Developed Our AI-Based SQL Rewrite System | Taboola.com - EN
https://www.taboola.com/engineering/rapido-sql-rewrite-system/

[6] Distributed count distinct vs. HyperLogLog in Postgres - Citus Data
https://www.citusdata.com/blog/2017/12/22/distributed-count-vs-hyperloglog/

[7] Consider Using [NOT] EXISTS Instead of [NOT] IN
https://dzone.com/articles/consider-using-not-exists-instead-of-not-in-subque

[9] [10] Advanced Postgres Performance Tips
https://thoughtbot.com/blog/advanced-postgres-performance-tips

[11] Partition predicate pushdown is not working with window functions · Issue #10352 · duckdb/duckdb · GitHub
https://github.com/duckdb/duckdb/issues/10352