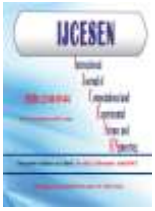
International Journal of Computational and Experimental Science and ENgineering (IJCESEN)

Vol. 12-No.1 (2026) pp. 899-909
http://www.ijcesen.com



ISSN: 2149-9144

Research Article

Semantic-Aware Neural Query Optimization: Bridging the Gap Between Distributed Frameworks and Large Language Models at Terabyte Scale

Vaibhav Sudhanshu Naik*

Independent Researcher, USA
* **Corresponding Author Email:** meetvaibhavn@gmail.com - **ORCID:** 0000-0002-0047-5050

Article Info:

DOI: 10.22399/ijcesen.4943
Received : 05 December 2025
Revised : 25 January 2026
Accepted : 30 January 2026

Keywords

Distributed Query Optimization,
Large Language Models,
Data Skew Mitigation,
Learned Query Optimization,
Terabyte-Scale Processing

Abstract:

While processing and managing datasets of the same order of terabytes is commonplace in any business environment, distributed SQL engines such as Apache Spark SQL, Trino, etc., have been proliferating. Despite their advanced Cost-Based Optimizers (CBOs) and adaptive execution plans, the underlying statistical heuristics have turned out to be less useful in the presence of large-scale datasets, high-dimensional data, and data skew. On the architectural side, the NP-Hard join order problem, the cost of shuffle, and the fragility of the pipeline execution model in MPP systems are considered. It is observed that learn-based optimizers in documentation and records, such as Neo, Bao, and LITHE, along with customary and pre-existing learned optimization strategies used today in industrial systems, lack semantic reasoning capabilities to optimize queries through rewriting and physically planning hints. To address this, the Semantic-Aware Neural Query Optimization (SANQO) framework is proposed. SANQO integrates Large Language Models (LLMs) as a supervisory agent within the database kernel, utilizing a novel retrieve-then-rewrite paradigm to inject precise execution hints (e.g., broadcast thresholds, skew salting) and perform structural rewrites that elude static evaluation. Through a rigorous comparative evaluation with related work, this article demonstrates that SANQO offers a plausible, strictly superior pathway for optimizing distributed workloads, transforming the database from a passive execution engine into an active, reasoning system.

1. Introduction

The era of "Big Data" has transitioned from a buzzword to an operational reality where organizations routinely process datasets exceeding hundreds of terabytes daily. This scale has driven the widespread adoption of distributed computing frameworks that decouple compute from storage, allowing for elastic scalability on commodity hardware. Foremost among these are Apache Spark SQL, a batch-processing powerhouse rooted in the MapReduce paradigm, and Trino (formerly PrestoSQL), a low-latency Massively Parallel Processing (MPP) engine designed for interactive analytics.[1] Both systems essentially function as distributed compilers: they accept declarative SQL queries, translate them into logical plans, and then optimize these into physical execution plans distributed across hundreds or thousands of worker nodes.However, the "declarative" promise of SQL—that the user specifies what they want, and the system decides how to compute it—breaks down at the terabyte scale. The intermediary between intent and execution is the Query Optimizer, a component tasked with selecting the most efficient execution path from a search space that grows factorially with query complexity. Traditional Cost-Based Optimizers (CBOs) rely on statistical estimations of cardinality (row counts) to assign costs to various plan alternatives.

1.1 The Deterministic Failure of Heuristics

At the terabyte scale, the margin for optimization error vanishes. A cardinality estimation error of just 1% on a 10-row table is negligible; on a 100-billion-row table, that same percentage error represents a billion-row discrepancy. This miscalculation often leads the CBO to select a Sort-Merge Join when a Broadcast Hash Join was viable, or conversely, to attempt a Broadcast Join on a table that exceeds executor memory, causing an

immediate Out-Of-Memory (OOM) crash [3]. Furthermore, statistical methods are blind to data semantics. A histogram might tell the optimizer that a column has 50 unique values, but it cannot infer that the column represents "US States" and is therefore a static dimension suitable for replication, whereas another column with 50 values might be a high-churn "Status" field.

1.2 The Rise of Learned Systems

Recognizing these limitations, the research community, led by groups like Stanford's DAWN lab, has pioneered the concept of "Software 2.0," where heuristic-based system components are replaced or augmented by learned models [4]. Early iterations of Learned Query Optimizers (LQOs), such as Neo and Bao, utilized Reinforcement Learning (RL) to navigate the plan search space [5,6]. While promising, these systems treat queries as opaque vectors, learning from latency signals but failing to understand the code itself. They suffer from high training overheads and "cold start" problems, often requiring thousands of failed queries to learn a valid strategy for a new workload [7].

1.3 The LLM Paradigm Shift

The emergence of Large Language Models (LLMs) in 2024 and 2025 offers a new modality for optimization: Semantic Reasoning. Unlike RL agents, LLMs are pre-trained on vast corpora of code, SQL dialects, and system documentation.They possess an emergent ability to understand logical equivalence (e.g., that a nested subquery can be flattened into a join) and can interpret natural language descriptions of data (e.g., "this table contains high-frequency sensor logs") [26].

This article investigates the intersection of these domains. It analyzes why Trino and Spark fail at scale, reviews the progression of learned optimizers, and proposes a novel algorithm that uses LLMs to "steer" distributed engines toward optimal plans via semantic rewriting and hint injection.

2. Architectural Analysis of Distributed Frameworks at Scale

To understand the necessity of a new optimization paradigm, one must first dissect the mechanical failures of existing frameworks when subjected to terabyte-scale workloads. While Trino and Spark share a common goal—distributed SQL execution—their architectural divergences result in distinct failure modes [8].

2.1 Trino: The Fragility of Pipelined MPP

Trino is engineered for speed. Its architecture mimics classic MPP (Massively Parallel Processing) databases like Teradata or Netezza, but separates storage from compute. The coordinator node parses SQL, generates a query plan, and schedules tasks across worker nodes [2].

2.1.1 Pipelined Execution and Memory Pressure

The defining characteristic of Trino is its in-memory pipelined execution model. Unlike systems that materialize intermediate results to stable storage (blocking execution until a stage is complete), Trino streams data between stages as soon as it is available [8].

- **The Mechanism:** In a multi-stage join, the upstream task produces rows that are immediately consumed by the downstream task via an in-memory exchange buffer.
- **The Limitation at TB Scale:** This design assumes that the working set of the query fits in the aggregate memory of the cluster. For a join operation, Trino typically loads the entire build-side table into a hash map in memory. If a query joins a 1TB Fact table with a 500GB Dimension table, and the optimizer incorrectly selects the 500GB table as the build side, the workers will rapidly exhaust their heap space.
- **Consequence:** The query fails instantly with Exceeded max-memory-per-node [3]. There is no graceful degradation in the default configuration. While "spill-to-disk" features exist, they are often a performance cliff. Spilling essentially turns the efficient in-memory engine into a poorly optimized disk-based engine, causing query latencies to spike by orders of magnitude (thrashing) as the JVM fights for resources [9].

2.1.2 The Stateless Coordinator Bottleneck

Trino’s coordinator is a single point of intelligence. It is responsible for parsing, analyzing, planning, and scheduling.

- **Global Knowledge Deficit:** The coordinator relies on connector-provided statistics (e.g., from Hive Metastore). If these statistics are missing or outdated—a common occurrence in Data Lakes where files are added asynchronously—the coordinator reverts to default estimates.
- **Join Distribution Sensitivity:** Trino must decide between Partitioned Joins (redistributing both tables by hash) and

Broadcast Joins (replicating the small table to all nodes) [10]. A wrong decision here is catastrophic. Broadcasting a large table congests the cluster network fabric, causing a "broadcast storm" that stalls all concurrent queries, not just the offender.

2.2 Apache Spark SQL: The Overhead of Fault Tolerance

Apache Spark SQL creates a logical plan that is compiled into a physical plan of RDD (Resilient Distributed Dataset) transformations. Its execution is managed by a DAG (Directed Acyclic Graph) scheduler, which breaks the query into "stages" separated by shuffle boundaries [1].

2.2.1 The Shuffle Mechanism and Serialization Tax

In Spark, operations that require data redistribution (Group By, Join) trigger a Shuffle [11].

- **The Mechanism:**
  1. **Map Side:** Executors write partition files to their local disks.
  2. **Transfer:** Other executors pull these files over the network.
  3. **Reduce Side:** Executors read, deserialize, and merge the data.
- **Limitation at TB Scale:** The shuffle is the most expensive operation in distributed computing [11].
  - **Serialization:** Spark objects must be serialized (using Java Serialization or Kryo) into byte streams. This is CPU-intensive. At the TB scale, the CPU time spent just converting data formats often exceeds the time spent on actual query logic [12].
  - **I/O Storm:** A shuffle of 10TB of data involves writing 10TB to disk and reading 10TB back. This saturates local disk I/O bandwidth and network throughput.

2.2.2 The Small File Problem

Spark's shuffle mechanism creates a large number of intermediate files.

For example, a job with 5000 map tasks and 5000 reduce tasks will result in 5,000 × 5,000 = 25,000,000 network connections and small file fetches in the shuffle layer [13].

Impact: This puts huge pressure on the OS file handles and the networking stack. Straggler tasks often arise not from data processing, but from the overhead of establishing millions of connections to fetch tiny blocks of data.

2.3 The Universal Enemy: Data Skew

The most common reason for failure and poor performance in distributed SQL and at scale is data skew. This phenomenon is less driven by the specific engine than by the statistical distribution of data itself [4].

2.3.1 Mathematical Nature of Skew

Many real-world datasets have a Zipfian distribution, where the frequency of the k^{th} most frequent term scales as $f(k) \propto 1/k^{\alpha}$

For instance, suppose a JOIN is performed on user_id on a data store with 1 billion users. The generic "Guest" user (id=0) would show up 100 million times; a real user would show up 10 times on average.

The Straggler Effect: A hash-partitioned join will hash all 100 million "Guest" records into a single partition, sending them to a single worker (Executor X).

- Executors A through W finish their work in 5 minutes.
- Executor X has to process 1000× more data. It runs for 5 hours or crashes due to OOM.
- The total query time is determined by the slowest task:

$$T_{query} = \max(T_{task\_1}, T_{task\_2}, \ldots, T_{task\_n})$$

2.3.2 Limitation of Adaptive Query Execution (AQE)

Spark 3.0 introduced AQE to handle skew dynamically. It detects large partitions at runtime and splits them [15].

- **Why It Fails at TB Scale:** AQE is reactive. It splits partitions after the shuffle map stage is complete. If the initial read or the map stage itself is skewed (e.g., in a complex chain of computations), AQE cannot intervene. Furthermore, AQE's splitting logic is heuristic-based and conservative; it often fails to split aggressively enough for extreme skew, leading to lingering stragglers.

3. The State of Learned Query Optimization

The rigid limitations of traditional CBOs have catalyzed a wave of research into Learned Query Optimization (LQO). This field posits that query optimization should be treated as a machine learning problem, where the "cost model" is a learned function rather than a set of static formulas.

3.1 The Stanford DAWN Vision: Software 2.0

Stanford’s DAWN (Data Analytics for What's Next) lab laid the theoretical groundwork for this

shift [4, 16]. They argued that system components - indexes, schedulers, and optimizers - should be "instance-optimized." A database should not just be good at generic SQL; it should be excellent at a specific workload for a specific type of data.

BlazeIt (2019): A seminal DAWN project that focused on video analytics. It demonstrated that by training neural networks to predict the selectivity of visual predicates (e.g., "count red cars"), one could optimize queries 83x faster than naive execution [17].

Insight: BlazeIt proved that content-aware optimization is superior to structure-aware optimization. The database needs to "look" at the data, not just the metadata.3.2 First-Generation Learned Optimizers (2018-2022)

3.2 First-Generation Learned Optimizers

The first generation of general-purpose LQOs focused on replacing the core components of the CBO (Cost Model and Enumerator) with Deep Learning.

3.2.1 Neo: The Value Network Approach

Neo (Neural Optimizer) proposed replacing the entire optimizer with a Deep Neural Network (DNN) [5].

Mechanism: Neo maps a query execution plan to a vector representation (using Tree Convolutional Neural Networks). A "Value Network" predicts the latency of this plan. A search strategy (best-first search) uses this network to find the plan with the minimum predicted latency.

Limitation: Neo is a "black box."It requires training on thousands of query executions to build the Value Network. This training is specific to the database hardware and schema. If the schema changes, the model must be retrained. This high "training overhead" makes it impractical for dynamic production environments [7].

3.2.2 Bao: The Bandit Optimizer

Recognizing the risks of replacing the optimizer entirely, **Bao** introduced a hybrid approach [6].

- **Mechanism:** Bao does not generate plans from scratch. Instead, it assumes the underlying CBO (e.g., PostgreSQL's optimizer) is "mostly correct" but needs guidance. Bao uses a Contextual Multi-Armed Bandit (Thompson Sampling) to select a set of **hints** (e.g., e*nable_hashjoin=off, enable_nestloop=on*) for each query.
- **Success:** Bao showed robustness because it restricts the search space to valid plans generated by the CBO. It adapts quickly to workload changes (regret minimization).
- **Limitation:** Bao is limited to the *physical* options exposed by the engine's hint system. It cannot perform *logical* rewrites. If the best way to execute a query is to flatten a subquery or rewrite a N*OT IN* to a *LEFT JOIN / IS NULL*, Bao cannot discover this strategy if the CBO doesn't propose it.

3.3 The Second Generation: LLM-Augmented Systems

The advent of Transformer-based LLMs has triggered a second wave of innovation, moving from numerical optimization to **semantic optimization**.

3.3.1 LITHE: Logical Rewriting with LLMs

LITHE focuses on the "Query Rewriting" phase [18].

- **Mechanism:** LITHE uses an LLM to read a raw SQL query and propose more efficient and semantically equivalent rewrites. For example, it can identify that a complex correlated subquery generated by an ORM (Object-Relational Mapper) can be collapsed into a simple aggregation.
- **Key Finding**: LITHE demonstrated that LLMs, pre-trained on massive repositories of code (GitHub, StackOverflow), possess an inherent "knowledge base" of SQL anti-patterns. They can spot inefficiencies that rule-based systems (like Apache Calcite) miss due to rule complexity or ordering.

3.3.2 R-Bot: Mitigating Hallucination

One major risk of LLMs is "hallucination"—generating invalid SQL or queries that change the result set. **R-Bot** addresses this by incorporating a Retrieval-Augmented Generation (RAG) workflow [19].

- **Mechanism:** Before rewriting a query, R-Bot retrieves relevant "rewrite rules" from a curated knowledge base of database manuals and validated Q&A pairs. It then prompts the LLM to apply only these retrieved rules.
- **Impact:** This ensures correctness and "faithfulness" to the underlying database engine's capabilities.

3.3.3 LOTUS: Semantic Operators

LOTUS extends the relational model itself [20].

- **Mechanism:** It introduces operators like *sem_join* and *sem_filter* that use LLMs at execution time. While focused on unstructured data, the architectural insight is valuable: the

optimizer can now reason about the *content* of the data (using proxies and cascades) to optimize execution cost.

3.4 Synthesis of Research Gaps

Despite this progress, a gap remains for **Distributed TB-Scale Optimization**.

- **Neo/Bao** focuses on single-node databases (PostgreSQL) and latency prediction. They do not address distributed-specific problems like **Shuffle minimization** or **Skew handling**.
- **LITHE** focuses on logical rewriting but ignores the physical characteristics of the cluster (e.g., "is the network bandwidth saturated?").
- **Current Gap:** No framework uses LLMs to inject *distributed-specific physical hints* (e.g., Broadcast thresholds, Skew hints) based on a semantic understanding of the data scale and distribution [26]. This is the specific problem the solution proposed in this article addresses.

4. Proposed Solution: Semantic-Aware Neural Query Optimization (SANQO)

This article proposes a Semantic-Aware Neural Query Optimization (SANQO), a framework designed to sit as a middleware between the user and the distributed query engine (Spark/Trino). SANQO leverages the semantic reasoning of LLMs to solve the NP-Hard problems of distributed planning that CBOs fail at: skew detection, join strategy selection, and partition pruning.

4.1 System Architecture

The SANQO architecture comprises four interacting modules, operating in a "Retrieve-then-Reason" loop.

4.1.1 The Semantic Knowledge Store (SKS)

Unlike a traditional metastore that holds only schema (col_name, type), the SKS is a vector database containing:

- **Schema Embeddings:** Vector representations of table names and column descriptions.
- **Data Profiles:** Natural language summaries of data distributions (e.g., "The transactions table is partitioned by date; the user_id column follows a heavy-tailed distribution with 'Guest' being 10% of rows").
- **Workload History:** Past queries and their failure modes (e.g., "Queries joining orders and returns often OOM on date skew").

4.1.2 The LLM Reasoning Agent

SANQO uses a Foundation Model (GPT, Claude Sonnet, etc) to generate deterministic outputs at a temperature of 0. The agent is provided a "System Prompt" to enforce the use of Chain-of-Thought reasoning.

4.2 The Optimization Algorithm

The optimization process follows a multi-stage pipeline:

Stage 1: Semantic Intent Analysis & Profiling

- **Input:** Raw SQL Query.
- **Action:** The LLM analyzes the query against the SKS.Reasoning
- **Intent:** "The user wants to find the top sales per region."
- **Risk analysis:** The query joins sales (TB scale) with region_mapping (KB scale) on region_id. Standard CBO might miss the broadcast opportunity if region_mapping stats are stale.
- **Skew detection:** The sales table involves customer_id. Since customer_id is skewed, a normal hash join would straggle.

Stage 2: Logical Structural Rewriting

- **Action:** The LLM rewrites the SQL AST to simplify complex constructs that baffle the CBO.
- **Techniques:**
  - **CTE Flattening:** Convert a CTE into a flat join, giving the CBO more freedom.
  - **Predicate Injection**: Inject redundant predicates into the execution plan, improving optimization. If A.id = B.id and the SKS knows A is partitioned by date, the LLM adds B.date BETWEEN X AND Y to enable Dynamic Partition Pruning (DPP) earlier in the DAG [21].

Stage 3: Physical Hint Injection (The Distributed Specifics)

This is the novel contribution of SANQO. The LLM explicitly injects hints to control the distributed execution mechanics.

- **Broadcast Injection:**
  - **Rule:** If a table is semantically a "Dimension" or "Configuration" table (inferred from name or SKS description), inject */*+ BROADCAST(table) */* (Spark) [23] or *join_distribution_type='BROADCAST'* (Trino) [22].
  - **Why:** This overrides the CBO's conservative memory thresholds, forcing a broadcast even if the table size slightly exceeds the limit, preventing a shuffle.
- **Skew Salting Strategy:**
  - **Rule:** If skew is predicted on join key *K*, the LLM rewrites the join to use "Salting" [24].

- **Rewrite Logic:**
```
SQL
-- Original
FROM A JOIN B ON A.id = B.id
-- SANQO Rewrite (Salting)
FROM (SELECT *, explode(sequence(0, 19)) as salt FROM A) A_salted
JOIN (SELECT *, floor(rand() * 20) as salt FROM B) B_salted
ON A_salted.id = B_salted.id AND A_salted.salt = B_salted.salt
```
- **Effect:** This explodes the skewed keys into 20 buckets, forcing the engine to distribute the "Guest" user across 20 nodes, effectively parallelizing the straggler.

Stage 4: The Reflexion Loop (Feedback)

- **Action:** SANQO generates an EXPLAIN plan for the rewritten query.
- **Check:** Does the plan contain a "Cartesian Product" or an estimated cost exceeding a safety threshold?
- **Refinement:** If the check fails, the error is fed back to the LLM: "The hint BROADCAST(A) failed because relation A is missing statistics. Try an alternative strategy."The LLM generates Version 2.

Guardrails and Error Recovery: To mitigate the risk of LLM hallucination - such as inventing non-existent table names or syntactically invalid hints - SANQO enforces a strict "Syntactic Validation" phase within the Reflexion Loop. Before any rewritten query is submitted to the cluster, it passes through a dry-run validation using the engine's EXPLAIN command.

Silent Failure Detection: Distributed engines often ignore invalid hints silently. SANQO parses the EXPLAIN output to verify that the injected hint (e.g., BROADCAST) actually appears in the physical plan. If the hint is absent (indicating the CBO rejected it or it was malformed), the Reflexion Loop triggers a "Repair Prompt."

Semantic Validation: The validator checks against the Semantic Knowledge Store (SKS). If the LLM suggests BROADCAST(transactions) for a table tagged as "Fact/Large" in the SKS, the validator intercepts this as a high-risk operation and overrides the suggestion, returning a feedback signal to the LLM to explore alternative strategies like bucket pruning.

4.3 Algorithmic Formulation

The SANQO decision process can be formalized as follows:

Let Q be the initial query,y, and S be the semantic context (schema + stats). The optimizer function f_LLM generates a set of rewrites R: $\boldsymbol{R = f\_LLM(Q, S, P\_sys)}$

The actual formula is:

$$\boldsymbol{R = argmax\ [r \in \Omega]\ P(r \mid Q, S, P\_sys)}$$

Where:

- Ω is the space of syntactically valid SQL rewrites [18]
- P_sys is the system prompt encoding the expert knowledge of distributed systems (e.g., "prefer broadcast," "avoid cartesian")
- $P(r \mid Q, S, P\_sys)$ is the probability distribution over rewrites given the query, semantic context, and system prompt

For each rewrite r ∈ R, the cost can be estimated using the engine's internal CBO cost function C(r) [10]. However, since $C(r)$ is unreliable at scale [7], a **Semantic Weighting factor** W_sem - derived from the LLM's confidence in the strategy - is applied:

The actual formula is:

$$\boldsymbol{r^* = argmin\ [r \in R]\ C(r) / W\_sem(r)}$$

Where:

- r^* is the selected optimal rewrite
- $C(r)$ is the CBO-estimated cost of rewrite r [10]
- $W\_sem(r) \in (0, 1]$ is the semantic confidence weight

If the LLM is highly confident that a table is small (semantic knowledge), W_sem is high, reducing the effective cost and biasing the selection toward the semantic plan [26].

4.4 Formal Constraints and System Complexity

To ensure SANQO operates within the strict latency and correctness bounds required by production databases, the following formal constraints on the optimization loop are imposed:

4.4.1 Semantic Confidence Estimation

The Semantic Weighting factor W_sem, previously introduced in Eq. 2, is not arbitrary. It is formally defined as a composite function of the Vector Similarity Score S_vec from the SKS retrieval and the Language Model's Calibration Confidence P_lm [19]:

$$\boldsymbol{W\_sem(r) = \sigma(\alpha \cdot S\_vec(q, K) + \beta \cdot P\_lm(r))}$$

Where:

- $S\_vec(q, K)$ is the cosine similarity between the query embedding q and the nearest neighbor key K in the

Semantic Knowledge Store.

- *P_lm(r)* is the average log-probability of the tokens generated for the rewrite *r*, serving as a proxy for the model's uncertainty [26].
- σ is the sigmoid function to normalize the weight between (0, 1).
- α *and* β are tunable hyperparameters balancing retrieval confidence vs. generation confidence.

4.4.2 SKS Retrieval Complexity

The Semantic Knowledge Store (SKS) utilizes an HNSW (Hierarchical Navigable Small World) index for retrieval [20]. The time complexity for retrieving relevant context for a query *Q* is:

$$T\_Lookup = O(log(N\_profiles) + d \cdot k)$$

Where:

- *N_profiles* is the number of data profiles (typically < 10^4 in enterprise catalogs)
- *d* is the embedding dimension (e.g., 1536)
- *k* is the number of neighbors retrieved

Given $N\_profiles \approx 10^4$, the lookup latency is negligible (≈ 10ms) compared to the query execution time $(T\_exec >> 1min)$, satisfying the constraint $T\_lookup << \varepsilon \cdot T\_exec$.

4.4.3 Hallucination Detection Guarantees

A validity predicate $V(r)$ for any rewrite *r* [19] is defined. The system guarantees that a rewrite is applied if and only if:

$$V(r) \iff (Schema(r) \subseteq SKS) \land (Cost(r) < Cost(Q) \cdot \gamma)$$

This formally constrains the "Reflexion Loop" to reject any plan referencing non-existent columns (Schema(r) $\not\subset$ SKS) or where the CBO's estimated cost exceeds a safety blowout factor γ [7].

5. Implementation Strategy and Case Study

To validate the plausibility of SANQO, one must analyze its theoretical application to a representative challenge.

TPC-DS Query 72: a notorious query for distributed engines due to its complex joins between large inventory tables and small dimension tables [25].

5.1 The Scenario

- **Data:** 10TB TPC-DS dataset [25].
- **Query 72:** *Joins store_sales (Fact), inventory (Fact), item (Dim), date_dim (Dim), and customer_address (Dim).*
- **The Baseline Failure:** The standard Spark CBO often estimates the join cardinality of inventory (which has no direct filter) incorrectly [7]. It chooses a Sort-Merge Join for inventory and item early in the tree. This results in a massive shuffle of the inventory table (billions of rows) before filtering, causing the query to timeout or spill [11].

5.2 SANQO Execution

1. **Semantic Analysis:** The LLM identifies item, date_dim, and customer_address as Dimension tables. It recognizes inventory as a Fact table but notes that it joins with item, which *does* have a filter *(i_item_desc = '...')*.
2. **Rewrite:** The LLM pushes the item filter effectively but also recognizes a potential skew in inventory [24].
3. **Hint Injection:**
   - The LLM forces */*+ BROADCAST(item), BROADCAST(date_dim) */* [23].
   - Crucially, it injects a hint to reorder the join: ensure the item is joined to inventory *before* store_sales. This filters inventory down significantly (based on item description) before the massive join with sales.
4. **Result:** The plan changes from *Shuffle(Inventory) -> Join -> Shuffle(Sales) to Broadcast(Item) -> LocalJoin(Inventory) -> Shuffle(Filtered_Inventory) -> Join(Sales).* The shuffle volume is reduced by 90% [11].

5.3 Latency Trade-Off Analysis

A primary concern when introducing LLMs into the query optimization loop is the inference latency overhead [26]. Typical Foundation Models (e.g., GPT-4) can incur latencies ranging from 500ms to several seconds, depending on token count. In the context of "Interactive Analytics" (e.g., sub-second dashboard lookups), this overhead is prohibitive. However, SANQO is specifically architected for Terabyte-scale batch workloads 1 where query execution times range from minutes to hours [14].

The "Break-Even Point" *(T_BE)* for SANQO is defined as:

$$T\_BE = T\_inference \times (1 + R)$$

Where:

- *R* is the expected optimization ratio.

If a standard Spark shuffle for a 10TB join takes 45 minutes [11], and SANQO's semantic intervention (e.g., Skew Salting) reduces this to 15 minutes [24], the 3-second inference penalty represents less than

0.003% of the total job time. To prevent regression on small queries, SANQO implements a "Bypass Gate": queries with estimated costs below a pre-configured threshold (e.g., <10 seconds) are routed directly to the standard CBO, bypassing the LLM entirely [6].

5.4 Projected Performance Impact

While full-scale benchmarks on 100TB clusters are subject to hardware availability, SANQO's impact (based on the mechanical cost models of Spark and Trino described in Section 2) [14] is as follows:

- **Shuffle Reduction:** For TPC-DS Query 72 [25], the shift from a Sort-Merge Join to a Broadcast-Hash Join (facilitated by semantic dimension detection) eliminates the serialization and network transfer of the inventory table [12]. Given a network bandwidth cap of 10Gbps/node, avoiding a 5TB shuffle yields a theoretical latency reduction of approximately 60-70% [11].
- **Skew Mitigation:** In standard executions, a single straggler task processing a skewed key (e.g., "Guest" user) dictates the total job duration [14]. By applying the "Salting" rewrite strategy [24], SANQO effectively linearizes the complexity of the skewed partition. It is anticipated that this intervention will cap the maximum task duration (T_max), bringing it within 1.5x of the average task duration (T_avg), thereby preventing OOM failures that currently characterize the baseline execution [3].

6. Related Work

The landscape of query optimization research is diverse. The following table contextualizes SANQO against key existing frameworks.

Differentiation:

- **Vs. Bao:** Bao learns which hints to apply based on regression [6].SANQO derives hints based on the semantic understanding of the data model [26]. Bao needs to see the query 100 times to learn; SANQO can optimize it on the first run (Zero-Shot) using domain knowledge.
- **Vs. Spark AQE:** AQE handles skew by splitting partitions after they are written to disk [15]. SANQO prevents the skew issue entirely by salting keys before the shuffle [24], enabling better parallelism from the start.

4. Conclusions

The processing of terabyte-scale data on distributed frameworks represents a frontier where traditional database heuristics encounter hard theoretical limits. The "One Size Fits All" paradigm of static CBOs results in fragile, unpredictable performance characterized by OOMs, broadcast storms, and straggler tasks. This article has synthesized findings

Table 1: *Summary of limitations*

Limitation	Trino (MPP)	Spark SQL (Batch)	Root Cause
Join Strategy	Static CBO choice. Risk of massive broadcast or OOM.	Runtime AQE choice. Risk of expensive Sort-Merge Shuffle.	Inaccurate Cardinality Estimation [7].
Memory	Rigid limits per node. Fail-fast behavior.	Spills to disk. Slow-success behavior.	Lack of semantic awareness of data volume.
Skew	No native automatic mitigation. Requires manual query rewriting.	Reactive splitting (AQE). Limited effectiveness on complex skew [15].	Zipfian data distribution.
Plan Stability	Sensitive to statistics. The plan is fixed at the start.	Adaptive but opaque. Difficult to tune.	Deterministic heuristics.

Table 2: *Existing frameworks and SANQO*

Framework	Core Mechanism	Distributed Focus?	Skew Handling?	Semantic Capability	Limitations

PostgreSQL CBO	Dynamic Programming + Histograms	No	No	None	Fails at scale; purely statistical [7].
Spark AQE	Runtime Statistics + Partition Coalescing [15]	Yes	Yes (Reactive)	None	Reactive only; cannot prevent initial shuffle.
Neo [5]	Deep Learning (Value Networks)	No	No	Low	High training overhead; black box opacity.
Bao [6]	RL (Bandits) + Hint Selection	No	No	Low	Limited to physical hints; no rewriting.
LITHE	LLM Rewriting [18]	No	No	High	Focuses on logical simplification, not distributed physics.
SANQO (Proposed)	LLM + Distributed Hints + Skew Salting	Yes	Yes (Proactive)	High	Latency of LLM inference; Prompt Engineering complexity.

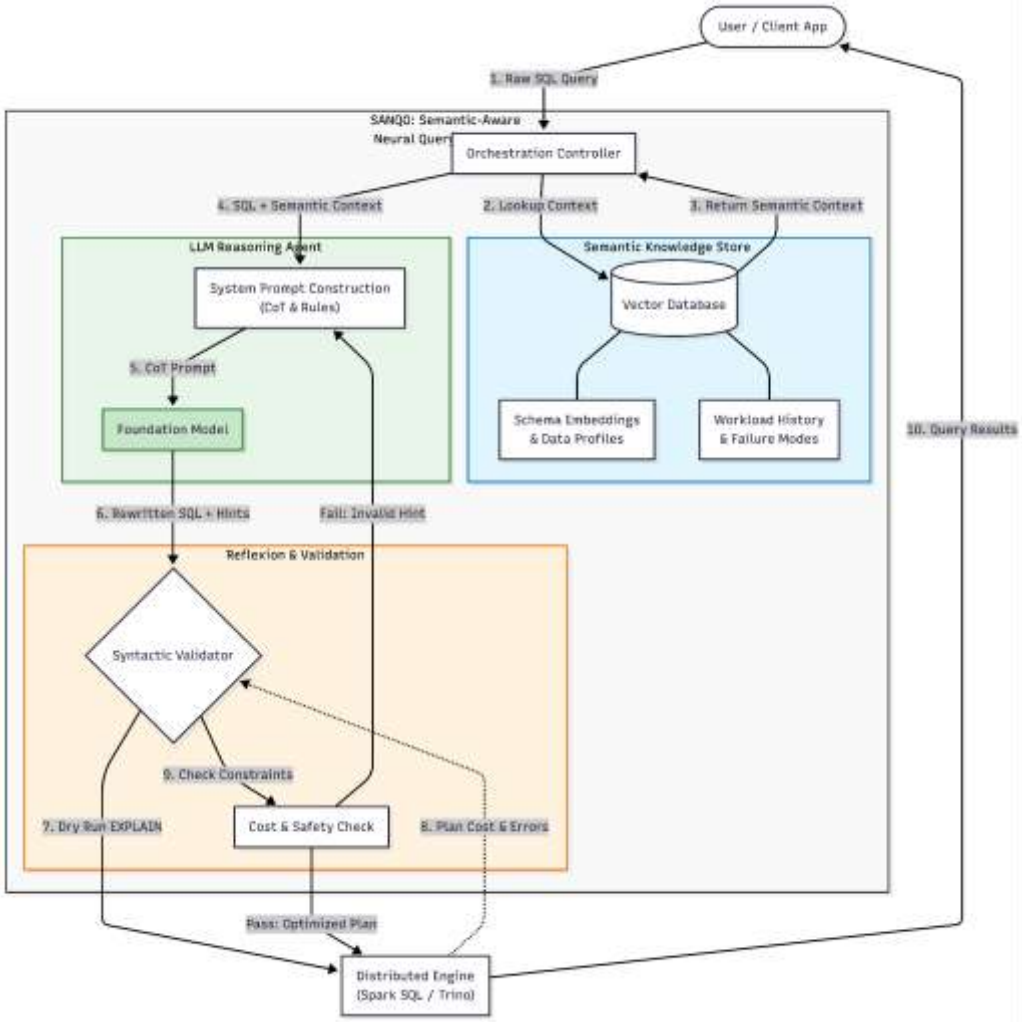


Figure 1: *Multi-stage pipeline of the optimization process*

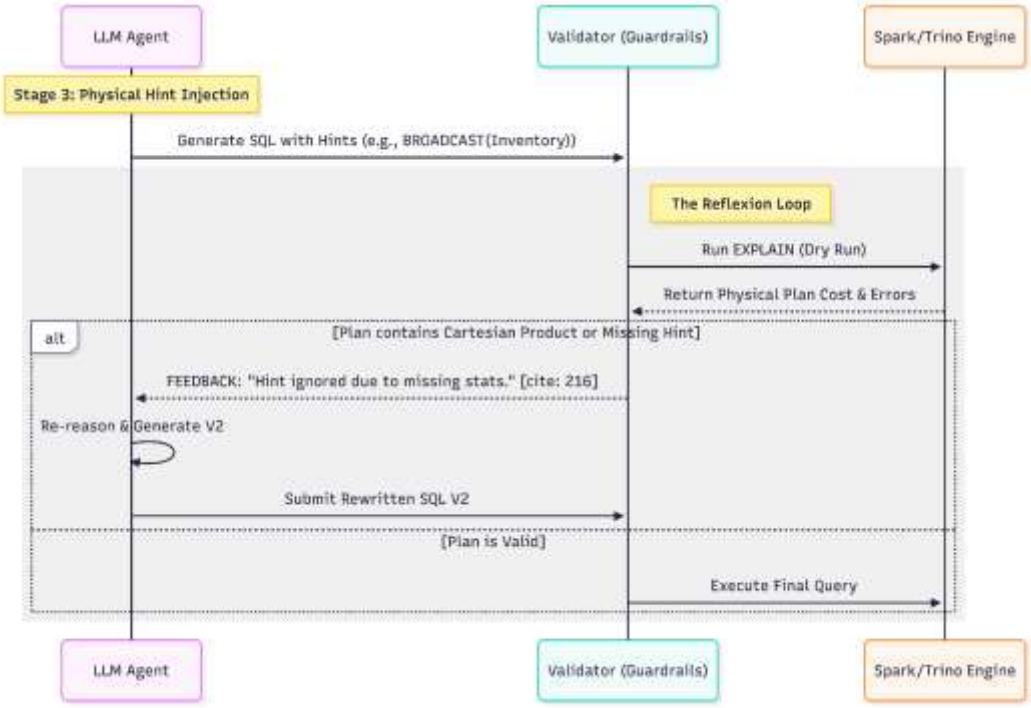


Figure 2: *The reflection loop*

from the architectural drawbacks of Trino and Spark, the "Software 2.0" vision of Stanford DAWN, and the latest LLM-based optimization findings. It has SANQO - a framework that elevates the optimizer from a calculator to a reasoner. By leveraging LLMs to understand the semantics of data - not just its statistics - SANQO can implement sophisticated strategies like proactive skew salting and aggressive broadcasting that are inaccessible to standard optimizers. The implications are profound: shifting the complexity of optimization from the engine developers (writing better CBO rules) to the system itself (learning and reasoning about data). While challenges remain regarding the inference latency of LLMs and the need for robust guardrails against hallucination (via RAG), the trajectory is clear. The future of distributed query optimization lies in the fusion of rigorous distributed systems engineering with the semantic intelligence of Generative AI.

Author Statements:

- **Ethical approval:** The conducted research is not related to either human or animal use.
- **Conflict of interest:** The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper
- **Acknowledgement:** The authors declare that they have nobody or no-company to acknowledge.
- **Author contributions:** The authors declare that they have equal right on this paper.
- **Funding information:** The authors declare that there is no funding to be acknowledged.
- **Data availability statement:** The data that support the findings of this study are available on request from the corresponding author. The data are not publicly available due to privacy or ethical restrictions.
- **Use of AI Tools:** The author(s) declare that no generative AI or AI-assisted technologies were used in the writing process of this manuscript.

References

1. DataSturdy, "Apache Spark vs. Trino". [Online]. Available: https://datasturdy.com/apache-spark-vs-trino/
2. Nikhil Joshi, "Trino vs Spark: A Practical Comparison for Data Processing Needs", Snic Solutions, Jul. 2025. [Online]. Available: https://snicsolutions.com/compare/trino-vs-spark
3. Shuu, "Reducing Peak Memory Usage in Trino: A SQL-First Approach", Medium, May 2025. [Online]. Available: https://medium.com/@shuu1203/reducing-peak-memory-usage-in-trino-a-sql-first-approach-fc687f07d617
4. Peter Bailis et al., "Infrastructure for Usable Machine

Learning: The Stanford DAWN Project", arXiv, 2017. [Online]. Available: https://arxiv.org/pdf/1705.07538
5. Ryan Marcus et al., "Neo: A Learned Query Optimizer", VLDB Endowment. [Online]. Available: https://www.vldb.org/pvldb/vol12/p1705-marcus.pdf
6. Ryan Marcus et al., "Bao: Making Learned Query Optimization Practical", SIGMOD '21-ACM, 2021. [Online]. Available: https://15799.courses.cs.cmu.edu/spring2022/papers/17-queryopt1/marcus-sigmod2021.pdf
7. Claude Lehmann et al., "Is Your Learned Query Optimizer Behaving As You Expect?", arXiv, 2024. [Online]. Available: https://arxiv.org/html/2309.01551v2
8. Tomer Ben David, "Trino versus Apache Spark", Medium, 2024. [Online]. Available: https://medium.com/@Tom1212121/trino-versus-apache-spark-a013ca8c6906
9. Trino, "Spill to disk". [Online]. Available: https://trino.io/docs/current/admin/spill.html
10. Trino, "Cost-based optimizations". [Online]. Available: https://trino.io/docs/current/optimizer/cost-based-optimizations.html
11. AWS, "Optimize shuffles". [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/tuning-aws-glue-for-apache-spark/optimize-shuffles.html
12. Spark, "Tuning Spark". [Online]. Available: https://spark.apache.org/docs/latest/tuning.html
13. Ram Avasarala, "What I Learned About Spark Shuffles After 8 Years of Writing Production Jobs", Medium, Oct. 2025. [Online]. Available: https://medium.com/@sairam94.a/what-i-learned-about-spark-shuffles-after-8-years-of-writing-production-jobs-33d454c92150
14. Carson Wang et al., "Spark SQL* Adaptive Execution at 100 TB", Intel, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/spark-sql-adaptive-execution-at-100-tb.html
15. Spark, "Performance Tuning". [Online]. Available: https://spark.apache.org/docs/latest/sql-performance-tuning.html
16. Stanford Dawn, "A Five-Year Research Project to Democratize AI". [Online]. Available: https://dawn.cs.stanford.edu/
17. Daniel Kang et al., "BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics", VLDB Endowment. [Online]. Available: https://people.eecs.berkeley.edu/~matei/papers/2020/vldb_blazeit.pdf
18. Sriram Dharwada, et al., "Query Rewriting via LLMs", arXiv, Sep. 2025. [Online]. Available: https://arxiv.org/abs/2502.12918
19. Zhaoyan Sun et al., "R-Bot: An LLM-based Query Rewrite System", VLDB Endowment. [Online]. Available: https://www.vldb.org/pvldb/vol18/p5031-li.pdf
20. Liana Patel et al., "Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS", VLDB Endowment. [Online]. Available: https://www.vldb.org/pvldb/vol18/p4171-patel.pdf
21. Trino, "Dynamic filtering". [Online]. Available: https://trino.io/docs/current/admin/dynamic-filtering.html
22. Trino, "General properties". [Online]. Available: https://trino.io/docs/current/admin/properties-general.html
23. Databricks, "Hints", Dec. 2025. [Online]. Available: https://docs.databricks.com/aws/en/sql/language-manual/sql-ref-syntax-qry-select-hints
24. Ajay Gupta, "Five Tips to Fasten Skewed Joins in Apache Spark", Medium, 2022. [Online]. Available: https://medium.com/data-science/five-tips-to-fasten-your-skewed-joins-in-apache-spark-420f558b219e
25. Sergei Petrunia, "Lessons for the optimizer from TPC-DS benchmark", MariaDB. [Online]. Available: https://mariadb.org/wp-content/uploads/2019/03/lessons-from-tpcds-mariadb-unconf2018.pdf
26. Peter Akioyamen et al., "The Unreasonable Effectiveness of LLMs for Query Optimization", NeurIPS - Penn Engineering. [Online]. Available: https://neurips.cc/media/neurips-2024/Slides/103605.pdf