Guided automated learning for query workload re-optimization

Guilherme Damasio[#◇], Vincent Corvinelli[$], Parke Godfrey[*◇], Piotr Mierzejewski[$],
Alex Mihaylov[#◇], Jaroslaw Szlichta[#◇], Calisto Zuzarte[$]
[#]Ontario Tech University, Canada [*]York University, Canada
[◇]IBM Centre for Advanced Studies, Canada, [$]IBM Ltd, Canada
guilherme.fetterdamasio@uoit.ca, vcorvine@ca.ibm.com, godfrey@yorku.ca, piotrm@ca.ibm.com,
alexandar.mihaylov@uoit.ca, jarek@uoit.ca, calisto@ca.ibm.com

ABSTRACT

Query optimization is a hallmark of database systems. When an SQL query runs more expensively than is viable or warranted, determination of the performance issues is usually performed manually in consultation with experts through the analysis of query's execution plan (QEP). However, this is an excessively time consuming, human error-prone, and costly process. GALO is a novel system that automates this process. The tool automatically learns recurring problem patterns in query plans over workloads in an offline learning phase, to build a knowledge base of plan-rewrite remedies. It then uses the knowledge base online to re-optimize queries often quite drastically.

GALO's knowledge base is built on RDF and SPARQL, W3C graph database standards, which is well suited for manipulating and querying over SQL query plans, which are graphs themselves. GALO acts as a third-tier of re-optimization, after query rewrite and cost-based optimization, as a *query plan rewrite*. For generality, the context of knowledge base problem patterns, including table and column names, is abstracted with canonical symbol labels. Since the knowledge base is not tied to the context of supplied QEPs, table and column names are matched automatically during the re-optimization phase. Thus, problem patterns learned over a particular query workload can be applied in other query workloads. GALO's knowledge base is also an invaluable tool for database experts to debug query performance issues by tracking to known issues and solutions as well as refining the optimizer with new tuned techniques by the development team. We demonstrate an experimental study of the effectiveness of our techniques over synthetic TPC-DS and real IBM client query workloads.

PVLDB Reference Format:
Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alexandar Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. Guided automated learning for query workload re-optimization. *PVLDB*, 12(12): 2010-2021, 2019.
DOI: https://doi.org/10.14778/3352063.3352120

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352120

1. INTRODUCTION

1.1 Motivation

As the complexity of queries, schemas, and database workloads spiral ever upward, the challenges in database systems have become severe. SQL queries nowadays often are generated by middelware tools instead of by SQL programmers [12]. Business-intelligence platforms such as IBM's Cognos have enabled organizations to systematically scale data analysis as never before. These benefits, however, come with a price. The generated SQL queries generated and run "behind the scenes" have essentially no limit on their complexity, often contain hundreds of algebraic operators and span thousands lines of code.

Query optimization has long been a hallmark of data warehouse systems, which has truly enabled the data-analysis revolution [13, 20]. However, there are cracks in the edifice; the complexity of (automatically-generated) queries and workloads is outpacing what database systems can perform efficiently. Database optimizers more often fail to pick best query plans. Research and development in query optimization is more vital today than it has ever had been, as people continue to address these new challenges [19].

Database vendors have made raw tools [5, 23] available to SQL programmers to troubleshoot performance problems for given queries, when the query optimizer fails to "do the right thing". Oracle offers the keyword *pragma* in its SQL, which can be used by the programmer to override decisions that the optimizer would make concerning, for example, choice of join algorithms and join order. (In truth, these pragma are suggestions to the optimizer.) Likewise, Microsoft SQL Server offers a similar mechanism via *hints*, which are embedded in the SQL query. IBM was reluctant to add a similar mechanism in DB2. Pragma and hints can go stale over time as the database's statistics change, yet they remain embedded in queries written in the past. IBM took a different approach: a *guideline document* (written in XML) can be submitted with a query to the optimizer. Like pragma and hints, the guidelines serve to sway the optimizer's choices in query planning.

The SQL programmer and database administrator (DBA) can analyze the queries from a workload with problematic performance, by profiling the query plans and execution traces, to troubleshoot performance issues. They then can override decisions in certain cases made by the database optimizer for these problem queries by using pragma, hints, and guidelines.

However, such performance debugging has become increasingly difficult with very complex queries and workloads. The causes of performance issues are, furthermore, often subtle. More time than ever is now spent by database system and optimization experts at the major database-vendor companies to help customers troubleshoot their workload performance problems. This troubleshooting is often manual and painstaking. Automatic tools are needed for vendor experts, SQL programmers and DBA's in the field for this workload debugging. However, existing tools lack the ability to impose the proper structure for the execution plans of queries [2, 7, 1].

This workload debugging also has been *ad hoc*. The lessons learned from the fix for one problematic query in one context—for a given database and system instance—are lost, to be rediscovered by others later. At IBM, our recently developed OptImatch system [9, 10] has been a successful effort towards addressing this. Experts feed problematic query-plan patterns and their resolutions into an OptImatch *knowledge base*. Thus, the expertise of problem resolutions is systematically stored, to be shared with and queried by others. Still, OptImatch knowledge base is built by experts manually by hand. The job of troubleshooting new performance issues remains exceedingly tedious and difficult. The knowledge base helps, though, experts to avoid re-solving previously solved issues, and to find similar patterns that can help with insights into the current issue.

1.2 Goals

In GALO, we extend ambitiously on the original goals of OptImatch. GALO's goals are threefold:
1. *automatic query problem determination*;
2. *query re-optimization*; and
3. *optimization evolution*.

Goal 1 is inherited from OptImatch. GALO significantly extends over OptImatch, however. The knowledge base is *automatically* "learned" rather than being manually constructed over the workloads. We use the RDF graph representation and the SPARQL language to impose the proper structure of execution plans for queries. GALO's architecture improves significantly on effectiveness and performance of the system, as discussed in Section 2.

Today's database optimizers are two stage: a *query-rewrite optimizer*; and a *cost-based optimizer* [21]. SQL offers many advantages for optimizing. The relational algebra offers many opportunities for re-ordering operations as much is associative and commutative. A query can be greatly rewritten as long as the variant is semantically equivalent to the original. Query rewrite applies well-known, well-tested transformations to an incoming query to "simplify" it, so that the resulting query plan will be more efficient.

The query-rewrite engine then passes the rewritten query to the cost-based optimizer. In cost-based optimization, statistics of the database and system parameters are used to make planning choices based on cost estimations. This generates a *query execution plan* (QEP). However, there might be "flaws" in the chosen query plan. Cost estimations may go awry. Unusual characteristics in the data and the query can circumvent the planning strategies as encoded in the optimizer.

In Goal 2, GALO offers a *third tier* of optimization, *query-plan* rewrite. Rules from GALO's knowledge base can be applied to the resulting query plan that remove known performance trouble spots. This is essentially an automation of the process done by hand by SQL programmers via pragma, hints, and guidelines. Furthermore, it applies all the acquired wisdom of performance debugging via the knowledge base, rather than ad hoc observations from the SQL programmers. Also, it is applied at the time the query is to be run, and not hard-coded into the SQL of the query itself (as with pragma).

One way this query-plan rewrite could proceed would be to "patch" the plan the cost-based optimizer produces by applying the matched rewrites to it. However, this could result in incompatibilities in the overall plan. Instead, GALO produces a guideline document with the chosen rewrites. Then the query with the guidelines is passed through the optimizer (the query rewrite and the cost-based tiers) again to ensure that statistics and operators are updated over subportions of the generated plan. This allows, just as in the case of an SQL query with pragma, hints, or guidelines, for the optimizer to generate a coherent query plan. Not all guidelines may be honored, as some may end up being incompatible within plan. The cost-based optimizer will use the most profitable ones. We call this *re-optimization*.

Goal 3 is long-term. GALO can be utilized by the performance optimization team to extract from the knowledge base those systemic issues for the optimizer, to learn and develop new rewrite rules for query rewrite and new optimization techniques and refinements for the cost-based optimizer. And these improvements are not merely academic; they arise directly from real-world workloads! GALO has been well received within IBM, and is proving to be a valuable tool both in company support and in database optimizer development.

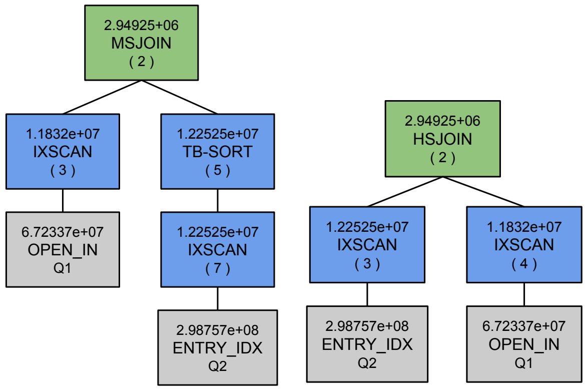
1.3 Real-world Example

Consider the portion of the tree of the query execution plan chosen by IBM DB2 as "optimal" shown in Figure 1a. The plan is comprised of a *merge join* (`MSJOIN`) between the `OPEN_IN` (`Q1`) and `ENTRY_IDX` (`Q2`) tables. Both are accessed via an *index scan* (`IXSCAN`). Each of the plan operators—e.g., `MSJOIN`, `HSJOIN`, `TBSCAN`, and `IXSCAN`—is referred in IBM DB2 as a low level plan operator (LOLEPOP). The topmost decimal number of each LOLEPOP corresponds to the optimizer's estimated cardinality, and the integer in parentheses represents the operator ID. For example, in Figure 1a, the LOLEPOP with the `MSJOIN` has an estimated cardinality of 2.94925e+06[1] and an operator ID of 2. For base tables, the topmost decimal value corresponds to the table's *cardinality* (the number of rows in the table), and the value under the table name corresponds to the *instance* of the table. For example, the table `ENTRY_IDX` has an estimated cardinality of 2.98757e+08, and a table instance value of `Q2`.

This pattern is an example of a real-life under-performing query from one of the IBM customers. The performance issue here hinges on the optimizer's join choice: the `MSJOIN` reads the table `ENTRY_IDX` through an `IXSCAN` (#7), and then performs a sort that is read by a table scan, `TB-SORT` (#5). The size of data into the sort and the number of pages that spill to the disk at runtime are large.

The chosen *fix* shown in Figure 1b changes the type of the join and the join order. This "changes" the `MSJOIN` to a

[1]Cardinalities are integer, of course; however, since this is an estimation, a decimal floating point is used to represent it.



(a) Plan by IBM DB2. (b) Plan by GALO.

Figure 1: IBM client query with a problematic join.

hash join (`HSJOIN`), and swaps the *outer* and *inner* tables as input to the join from `OPEN_IN` on the left and `ENTRY_IDX` on the right to `ENTRY_IDX` on the left and `OPEN_IN` on the right. While the `HSJOIN` spills pages into the disk at runtime too, the amount is significantly smaller.

This plan rewrite reduced the query runtime from nine hours to just five minutes! However, as it is evident from the rewrite in Figure 1, the steps required to fix existing performance issues are not always intuitive or simple. IBM experts report that even more complex patterns / rewrites exist that are more daunting, which can take days to be found and resolved.

1.4 Contributions

We developed the GALO system, which can serve as a third tier of optimization by rewriting problematic portions of query plans to result often in dramatically increased performance. Our main contributions are as follows.

1. **The Knowledge Base.** First is GALO's *knowledge base*, an innovative and powerful representation for storing, manipulating, and querying SQL query-plan patterns. GALO's *transformation engine* is responsible for mapping SQL queries and query plans to the knowledge base's RDF format, and to the SPARQL queries used to query the knowledge base.
2. **The Learning Engine.** Second is the *learning engine*, which is used *offline* to populate the knowledge base. It analyzes large and complex SQL queries in the workload, and segments them into sub-queries. The set of sub-queries is then broadened; for each sub-query, the values of the query's predicates are varied to result in different *reduction factors* (and, hence, result cardinalities). Then, for each sub-query from this broadened set, the query plan that the optimizer produces is compared against competing plans found using DB2's Random Plan Generator. Whenever a competing plan is found that performs significantly better than the optimizer's, it flags the pair as a potential *rewrite* (a problematic plan pattern and its guideline solution). For *abstraction*, the table and column names in the plans are replaced by canonical symbol labels.
3. **The Matching Engine.** Third is the *matching engine*, which is employed *online* to re-optimize the query plans of incoming queries by querying the knowledge base (via SPARQL queries) to find matching plan rewrites. Since the knowledge base's rewrites are abstracted (with canonical symbol labels for tables and attributes), a query with the similar sub-structure and characteristics can match a problem pattern of a rewrite that had been discovered during learning over a different query, even from a different query workload. The SPARQL queries generated for matching into the knowledge base support naturally this abstraction. SPARQL node-binding variables match to the canonical names.
4. **Experiments.**
   (a) We demonstrate experimentally dramatic query performance improvement and scalability of our solution over the TPC-DS benchmark and real-world IBM customer query workloads. We also show that problem patterns learned over one query workload are re-used when re-optimizing queries in other workloads.
   (b) We quantify the benefits of our automatic approach against manual diagnosis. A collaborative study illustrates that the system is able to perform more effectively than IBM experts by providing more optimized solutions for problematic queries, while saving a significant amount of time to analyze QEP's.

In Section 2, we overview GALO's architecture . In Section 3, we describe in detail the three primary components as discussed above, the knowledge base, the learning engine, and the matching engine. In Section 4, we provide a comprehensive experimental evaluation. In Section 5, we discuss the related work and we conclude in Section 6.

2. GALO SYSTEM OVERVIEW

GALO is an automated system to improve SQL workload performance. We consider a *workload* here to be a populated database with a requisite schema and a collection of SQL queries that are periodically executed on a given database system instance. GALO *profiles* the workloads *offline* to construct a *knowledge base* which captures performance issues (that have resolutions) from the queries in the workloads. When the workload is executed (e.g., periodically in a data warehouse), GALO acts as a *third stage* of re-optimization by applying *rewrites* from the knowledge base (KB) to the query plans *online* to improve performance.

GALO extends upon our OptImatch system [9, 10]. As the "second generation" of OptImatch, GALO is used within IBM to populate automatically a general knowledge base that tracks query plan issues (as opposed to building such manually, as was done in OptImatch). The knowledge base is used to resolve customers' workload performance issues. GALO is also being used as a resource for the IBM DB2 engine team for evolving the DB2 optimizer. Rewrites in the knowledge base (improved query plans) can be extracted and generalized to inform the optimizer team, where optimization rules should be added and refined.

GALO's system architecture is illustrated in Figure 2. The system is comprised of

1. a *transformation engine*,
2. a *learning engine*, and
3. a *matching engine*.

The back-end of GALO is written in Java. The front-end is a web-based, interactive interface written with JavaScript libraries.

The transformation engine is the primary interface to go from SQL queries and query-execution plans into the knowledge base and back. The knowledge base is represented in

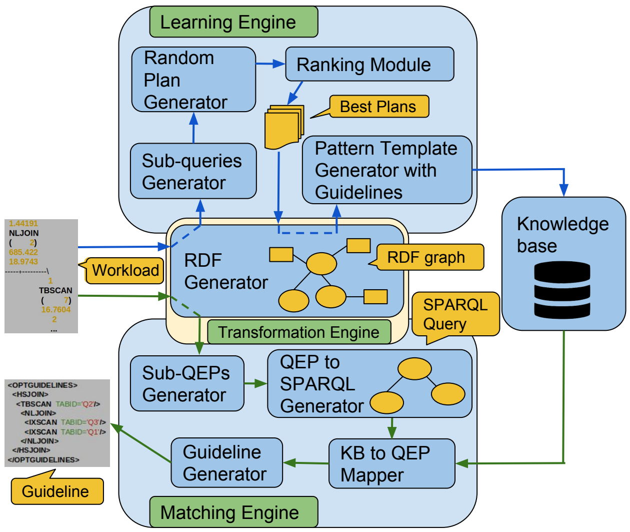


Figure 2: System architecture of GALO.

RDF and interacted with (*queried via*) SPARQL. The *learning engine* is used *offline* to populate the knowledge base with discovered rewrites. The *matching engine* is used *online* to match rewrites to query plans of SQL queries from the workload queued for execution for the purpose of re-optimization.

There are, thus, two *workflows* for GALO: *offline learning* and *online re-optimization*. In Figure 2, *offline learning* workflow is on the top, through the transformation and learning engines, for updating the knowledge base. The *online re-optimization* workflow is on the bottom, through the transformation and matching engines, employing the knowledge base.

3. MODULES

We detail the *knowledge base* (along with the *transformation engine*), the *learning engine*, and the *matching engine*, each in turn.

3.1 Knowledge Base

A *query execution plan* (QEP)—or simply a *query plan*, for short—is the executable plan constructed by the query optimizer for an SQL query to be evaluated at runtime. Within IBM DB2, query plans are represented in the *query graph model* (QGM). An SQL query is parsed into a QGM representation; such a plan within IBM is referred to as a "QGM". That QGM is then rewritten by DB2's query rewrite engine, which applies general heuristic transformations to the "query" (the QGM) known to generally simplify the query for purposes of evaluation. The resulting QGM is then passed to DB2's cost-based optimizer, which annotates the QGM to a full-fledged query plan, which constitutes a query execution plan.

A QGM can be read as a diagnostic file as produced by the IBM DB2 optimizer. The QGM profiles the *access paths* chosen by the optimizer, the chosen join types (e.g., sort-merge, index nested loop), the join order, and index usage. The QGM is represented as a compact graph structure. (Hence the name, *query graph model*.) Each LOLEPOP is described in detailed textual blocks identified by ID. Figure 1 depicts a portion of the QGM file that results for a real-world SQL query from an IBM client. Each node of the tree represents an indivisible operator (LOLEPOP), along with its associated estimated costs.

As QGM's are essentially graphs, representing them as such is natural. For building, maintaining, and accessing a knowledge base of query plans, we want a flexible graph representation, and a powerful, general "API" for accessing and maintaining the knowledge base. We choose then for the representation of the knowledge base, the Resource Description Framework (RDF). RDF's corresponding SPARQL query language provides the means to query and update the knowledge base. Matching to (sub-)query plans as stored in RDF in the knowledge base requires recursive path matching in the graph; such *regular path queries* (called *property paths* in SPARQL) are part of SPARQL 1.1, and are a fundamental part of the query language.

The transformation engine is GALO's general tool to translate QGM's and SQL queries into RDF graphs (and to translate back). An RDF graph is (conceptually) comprised of triples: *subject* (resource); *predicate* (property or relationship); and *object* (value or resource). As such, an RDF triple describes an "edge" in the graph from the vertex *source* to the "vertex" *object* and labeled as *predicate*. RDF also allows for the object of a triple to be a *value* instead of another node.

RDF statements can describe characteristics of subjects via predicates and values. The relationships between LOLEPOP's of a QGM can be thus modeled. The entities and characteristics of the QGM are mapped into resources, capturing the properties and relationships between them. The resulting RDF graph is a full transformation of the text-based QGM.

Once modeled as an RDF graph, SPARQL queries can be used to find matches in the graph. SPARQL's recursive capabilities via property paths let us search for loosely connected child operators (separated by other operators), and to match patterns that appear multiple times throughout the same QGM. This approach is, of course, also more efficient than ad-hoc usage of UNIX tools, such as *regex* and *grep*, that experts still use to search QGM files themselves.

GALO uses the Apache Jena RDF API to map the QGM into an RDF graph. Jena RDF API is a Java framework that can be used for the creation and manipulation of RDF graphs [17]. Jena is a popular option in the domain. It is an open-source framework for building linked data, it has an API for building RDF graphs, and it natively supports triple store servers such as Jena's Fuseki. For example, consider below a portion of the RDF graph as translated from the QGM presented in Fig. 1a. The statement

<http://galo/qep/pop/2><http://galo/qep/property/ hasPopType>NLJOIN

represents the subject (LOLEPOP) with ID #2, the predicate *hasPopType*, and the object NLJOIN. This subject also contains additional properties, such as the estimated cardinality, containing the value of 2949250.

<http://galo/qep/pop/2><http://galo/qep/property/ hasEstimateCardinality> "2949250"

This LOLEPOP connects to another LOLEPOP with ID #3 as its outer input stream.

<http://galo/qep/pop/2><http://galo/qep/property/ hasOuterInputStream><http://galo/qep/pop/3>

```
SELECT  i_item_desc ,i_category ,i_class ,i_current_price
FROM     web_sales, item, date_dim
WHERE   ws_item_sk = i_item_sk and
         i_category = 'Jewelry' and
         ws_sold_date_sk = d_date_sk and
         d_date = '2016-01-02'
```

(a) Sample query with joins and predicates.

```
SELECT  i_item_desc ,i_category ,i_class ,i_current_price
FROM     web_sales, item
WHERE   ws_item_sk = i_item_sk and
         i_category = 'Jewelry'
```

(b) Generated sub-query.

Figure 3: Sub-queries generation process.

3.2 Learning Engine

During the offline learning process, large and complex SQL queries from the workload are *segmented* into sub-queries. Random plans are generated over and benchmarked (via runtime performance) against the optimizer's chosen plans. When better random plans are discovered, they are ranked to determine the best QGM for each sub-query. Finally, the best are abstracted into rewrites, template patterns, to be stored in the knowledge base.

The learning engine is run *offline* inside the organization, when the resources over the systems are not in use, or when load is low. This includes nights and other non-peak hours, such as weekends and holidays. We used several machines inside IBM during non-peak hours to improve scalability by paralleling the computation.

Sub-query Generation. The learning engine is responsible for populating the knowledge base with problem pattern templates and their counterpart recommendations. Large SQL queries are decomposed into smaller parts corresponding to sub-queries to find problematic patterns that can be applied over the query workloads for re-optimization (discussed in the next section).

From a given RDF-based QGM, all SQL sub-queries are auto-generated up to a predefined size threshold (number of joins). A sub-query projects the join and local predicates from the original query that are applicable to the sub-query's selected tables. An example of the sub-query generation process is illustrated in Figure 3. Figure 3a shows the original SQL query, which consists of a three-way join between TPC-DS benchmark schema tables *web_sales*, *item* and *date_dim*. Figure 3b represents one of the sub-queries, a two-way join between tables *web_sales* and *item*.

The system produces potential problem-pattern *templates* from sub-queries by generating over predicates' property ranges with various cardinalities. Property ranges are generated by sampling the database, and are used to establish problem-pattern templates with the same best plan within lower and upper-bound cardinalities. This precaution is to ensure that problem patterns discovered over one query can be used to match other queries with different contexts of table and attribute names, but with the same sub-structure. For example, to create the property ranges for the sub-query in Fig. 3b, the *i_category* attribute in the *WHERE* predicate is sampled from the TPC-DS database to find varying cardinalities. For isntance, the predicate "*i_category is NULL*" returns 1,949 rows and "*i_category = 'Music'*" returns 74,426 rows.

The learning engine is designed to operate on top of dynamic data environments with changing statistics. As data change, the lower and upper-bound cardinalities for problem patterns can be updated over the time to account for cardinalities not observed before.

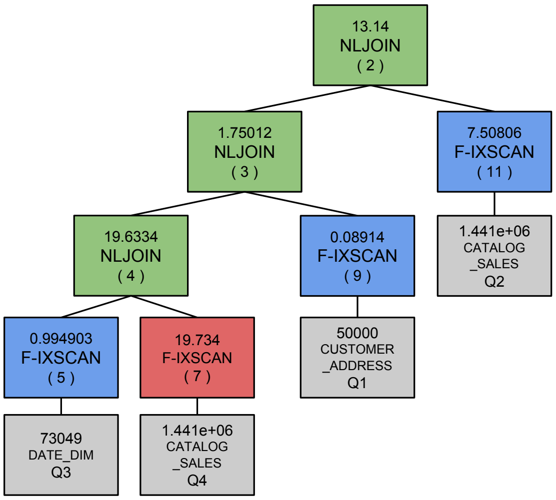
The Ranking Process. For each of the sub-queries, alternative QGM's are produced via the Random Plan Generator (a tool available inside IBM DB2). Alternative plans are compared against the QGM chosen by the optimizer as "optimal". As the cost estimates used during optimization are not always accurate with respect to what is observed at runtime, the runtime statistics are obtained by executing the alternative QGMs via DB2's `db2batch` utility tool. The ranking objective is to determine the best QGM for each sub-query within the predicate property ranges. Ultimately, if multiple competing plans were found to be better for a given sub-query, the best is chosen as a rewrite to add to the knowledge base.

Each QGM is run multiple times to obtain an accurate baseline cost, to remove noise related to the server or network load. The ranking process uses K-means clustering to remove outliers based on elapsed time. The clustering algorithm divides QGM's into two clusters: *prospective* and *anomaly*. QGM's in the prospective cluster are then considered, while those in the anomaly cluster are ignored. In the case of ties, the system considers other features as a tie breaker. These are measures of other resource usages, such as buffer pool data logical reads and physical reads, total CPU time usage, and shared sort-heap high-water mark.

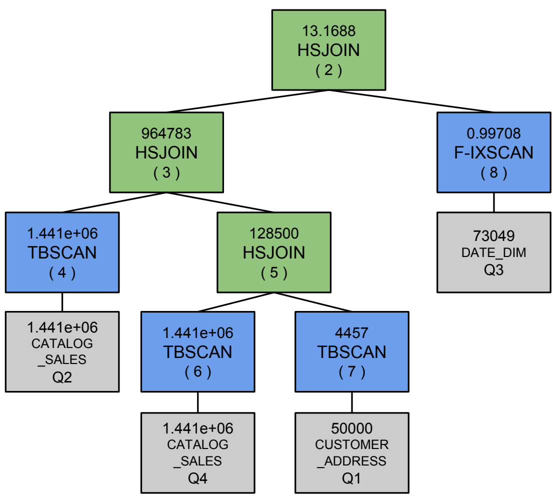
A problematic portion of QGM as chosen by the optimizer from the TPC-DS workload query is shown in Figure 4a. At runtime, the `F-IXSCAN` (#7) suffers from excessive random I/O reads. There is a *flooding* problem. This is a consequence of a poorly clustered index used to access the `CATALOG_SALES` table instance `Q4`, causing pages to be loaded into the buffer pool as usual, however, then being overwritten by other pages subsequently loaded. When a replaced page needs to be read again, it is subsequently loaded back into the buffer pool. This adds significant I/O's. This results in a poorly performing `NLJOIN` (#4) when joining the problematic `F-IXSCAN` (#7) with the `F-IXSCAN` (#5) over the `DATE_DIM` table instance `Q3`. This overhead is propagated upward into the next `NLJOIN` (#2) and operators to follow, causing further performance issues.

Early occurrence performance issues like this must be addressed as they affect the whole QGM. These are the types of challenges that experts encounter. With a sparsity of accessible solutions, automated methods become that much more important. GALO finds a solution to the problem pattern in Fig. 4b. The discovered solution applies a hash-join bloom filter in the `HSJOIN` (#2). A bloom filter is a space-efficient, probabilistic data structure to test whether an element is a member of a set by hashing the values and performing a bit comparison between them [3]. False positives can occur; however, false negatives never occur. A bloom filter can filter so whole partitions never need to be read, as we know by a filter miss, nothing in the partition can match. In the better query plan, the hash join creates a bitmap from the inner input. This bitmap is used as a bloom filter lookup for the join, to avoid hash-table probes for outer tuples that never can match. This results in a drastically faster execution plan.

Knowledge-base Generation. Detected query problem



(a) Plan obtained by the optimizer.



(b) Faster plan found by GALO.

Figure 4: Hash-Join Bloom Filter Problem Pattern.

patterns are transformed into *templates* to be saved in the knowledge-base RDF graph. This is a critical abstraction step that enables different queries with varying tables and predicates later to match to patterns in the knowledge base. Table and column names are replaced by the canonical symbol labels in the QGM. When SPARQL queries are generated for the matching for online re-optimization, the SPARQL node-binding variables will match to these. Thus, queries with the same sub-structure and characteristics, but with different table and attribute names, are matched against the same problem pattern template. This assures that problem patterns usability is not limited to a specific query or query workload. Over time, a *unified knowledge base* of problem patterns and guidelines learned over various query workloads is created. Our experiments in Section 4.2 (Exp–2) demonstrate that, in practice, problem patterns overlap significantly across various query workloads. A template is generated over the predicate property ranges with the same best plan by sampling the database with various cardinalities, to establish the lower and upper bound for properties.

The generated resources are keyed by the IDs of the LOLEPOPs in the QGM, different resources could potentially have name collisions between their problem-pattern templates in the knowledge base. To rectify this, each resource is anonymized by generating a unique random identifier.

The upper- and lower-bound values are each stored in their own respective tags in the predicate. For instance, the upper-bound value for the **hasCardinality** property is stored as **hasHigherCardinality**, and the lower-bound value is stored as **hasLowerCardinality**. Consider a portion of the RDF graph corresponding to the hash-join bloom filter pattern from Figure 4. Our system was able to determine that LOLEPOP #5 has a cardinality lower bound of 19,771 and upper bound of 128,500.

<http://galo/qep/pop/5><http://galo/qep/property/hasLowerCardinality> "19771"
<http://galo/qep/pop/5><http://galo/qep/property/hasHigherCardinality> "128500"

This problem pattern template dictates that any QGM that falls in the given range and matches the rest of the structure should be re-optimized.

The knowledge base is housed in an Apache Jena Fuseki SPARQL server. Fuseki is a SPARQL end-point accessible via HTTP protocols. This provides a REST (Representational State Transfer) API for querying the knowledge-base graph on the server. We opted for this service as it is integrated with TDB (Native Triple Store), a Jena component for RDF storage and querying. While TDB can be used as a RDF storage on a single machine, Fuseki has parallelism built in, enabling multiple requests to be performed concurrently. It provides a robust, transactional, and persistent storage layer.

The recommended replacement patterns for corresponding "malicious" problem pattern templates are stored in the knowledge base as *guidelines*. A guideline document is represented as an XML document. It imposes characteristics on the plan during the cost-based phase of optimization, such as the join methods (e.g., hash-join or merge-join), join order (enforced by the order of the XML tags), and access methods (e.g, index scan). Join tags in the guidelines require two child elements: the first corresponding to the outer input; and the second to the inner input of the join. A plan re-optimization guideline document does not necessarily specify all aspects of the execution decisions. Unspecified aspects of the execution plan will default to being chosen by the optimizer in a cost-based fashion. A guideline document can be generated by the matching engine, then, to be provided with the SQL query back to the optimizer for "re-optimization".[2]

Figure 5 illustrates the XML guideline generated for the QGM in Figure 4b. The `HSJOIN` tags on Lines 2, 3 and 5, correspond to LOLEPOP operator IDs #2, #3 and #5 in Figure 4b, respectively. The `HSJOIN` element on Line 5 contains two child elements. The first element indicates that table `Q4` is an outer input of the join accessed by `TBSCAN`. Similarly, the second element indicates that table Q1 is an inner input of the join, also to be accessed with `TBSCAN`. The `TABID` attribute specifies the table reference to which the access should be applied. The attribute's target table reference is identified by its qualifier name from the QGM.

[2]Note that a guideline, in truth, is a strong suggestion to the optimizer. A guideline will not be used if other previous employed guidelines lead to a (partial) query plan in which the guideline in question is no longer applicable.

```
1   <OPTGUIDELINES>
2       <HSJOIN>
3           <HSJOIN>
4               <TBSCAN TABID='Q2'/>
5               <HSJOIN>
6                   <TBSCAN TABID='Q4'/>
7                   <TBSCAN TABID='Q1'/>
8               </HSJOIN>
9           </HSJOIN>
10          <IXSCAN TABID='Q3'
11                INDEX='"D_DATE_SK"'/>
12      </HSJOIN>
13  </OPTGUIDELINES>
```

Figure 5: Guideline generated for plan in Figure 4b.

Alternatively, the TABLE attribute can be used instead, specifying the fully qualified table name. The children of a join element does not necessarily have to be an accessor to tables, however. It can instead be the other join element. The HSJOIN on Line 2 depicts this exact scenario. The outer input child element (Line 3) is another HSJOIN, and the inner input element an IXSCAN (Line 10). The latter specifies that the optimizer should use the D_DATE_SK index to access the Q3 table. (The optional INDEX attribute specifies the desired index to be used in the plan).

3.3 Matching Engine

Querying Knowledge Base. SPARQL is a recursive acronym for SPARQL Protocol and RDF Query Language. As the acronym suggests, the languages is able to retrieve data stored in the RDF format.[3] A SPARQL query consists of a set of triple patterns similar to RDF triples. In the query, each of the *subject*, *predicate*, and *object* may be a variable. As discussed before, SPARQL has many language features that are quite beneficial for GALO's tasks. Since our templates are stored as RDF graphs in the knowledge base, querying these by SPARQL provides an efficient way to match problematic patterns and to retrieve the corresponding recommended patterns.

At runtime, a potentially large and complex SQL query to be re-optimized is segmented into sub-queries (in the similar fashion as queries are in the learning phase). The transformation engine is used to translate the query—represented as an initial QGM by DB2 after the query is parsed—into RDF and, there, segmented. The transformation engine then rewrites the RDF's segments into SPARQL queries, with the necessary characteristics to match against the RDF problem pattern templates in the knowledge base. (Note this is akin to a *query by example*.) The resulting SPARQL query is composed of two parts: a SELECT clause, which defines the variables to be retrieved; and a WHERE clause enumerating the properties to match. Variable names in SPARQL are prefixed by "?". To facilitate this SPARQL query generation, we introduce handlers to generate automatically the variable names. We define three types of handler variables: *result*, *internal*, and *relationship* handlers.

A *result handler* names the results retrieved from the query. It is composed of the name pop and the ID of the LOLEPOP (or the name of the table instance for an access path). This is used in the SELECT statement for returning the resource, and inside the WHERE statement for creating

[3]SPARQL and RDF are W3C standards which were originally developed for *semantic web*.

```
PREFIX predURI: <http://optimatch/qep/property/>
SELECT ?pop_Q3 ?pop_6 ... ?pop_4
WHERE {
 ?pop_Q3 predURI:hasLowerRowSize ?ih1 .
FILTER ( ?ih1 <= 8) .
 ?pop_Q3 predURI:hasHigherRowSize ?ih2 .
   FILTER ( ?ih2 >= 8) .
 ?pop_Q3 predURI:hasLowerFPages ?ih3 .
   FILTER ( ?ih3 <= 656) .
 ?pop_Q3 predURI:hasHigherFPages ?ih4 .
FILTER ( ?ih4 >= 656) ...
  ?pop_6 predURI:hasLowerCardinality ?ih15 .
   FILTER ( ?ih15 <= 1) .
  ?pop_6 predURI:hasHigherCardinality ?ih16 .
   FILTER ( ?ih16 >= 1) .
  ?pop_4 predURI:hasLowerCardinality ?ih29 .
   FILTER ( ?ih29 <= 1372) .
  ?pop_4 predURI:hasHigherCardinality ?ih30 .
    FILTER ( ?ih30 >= 1372) .
    FILTER (STR(?pop_6) > STR(?pop_8)) ...
 ?pop_Q3 predURI:hasOutputStream ?pop_6 .  ?
pop_6 predURI:hasOutputStream ?pop_4 .}
```

Figure 6: SPARQL query for pattern in Fig 4a.

properties of a pattern. Figure 6 illustrates a portion of an auto-generated SPARQL query for the problem pattern of the QGM from Figure 4a. Here, ?pop_Q3 and ?pop_6 correspond to the TABLE 1 table instance Q4 and the IXSCAN under the FETCH #7, respectively. The result handler ?pop_Q3 is a resource returned back via SELECT that is also used in the WHERE clause to identify the characteristics of this resource, such as the row size (obtained by adding the predicate hasLowerRowSize and hasHigherRowSize).

An *internal handler* is used to aid in the stating of properties such as filtering. We name it by "ih" (for *internal handler*) appended with a sequential identifier. In Fig. 6, an internal handler is used to filter values for the pop_4 cardinalities, first by associating it to the resource pop_4 ("?pop_4 predURI:hasLowerCardinality ?ih29"), and then within a FILTER clause ("FILTER (?ih29 $\leq$ 1372)").

For each property, the SPARQL query ensures that the values are within the problem-pattern template range. In Figure 6, the property hasLowerCardinality is used to check the lower bound of the cardinality. The higher bound is checked by hasHigherCardinality, accordingly. The FILTER statement is used to enforce the uniqueness of each resource via a distinct resource ID. LOLEPOPs #6 and #8 are assured distinct by applying the following filter:

"FILTER(STR(?pop_6) > STR(?pop_8))".

A *relationship handler* establishes a connection between nodes. It is denoted by a *result handler* in conjunction with the property hasOutputStream. In Figure 6, to connect the FETCH IXSCAN #6 to the NLJOIN #4 (from Figure 4a), the tool generates in the WHERE clause a relationship statement

"?pop_6 predURI:hasOutputStream ?pop_4".

Plan Transformation. The QGM generated by the optimizer is modified by matching RDF problem pattern templates. The matches are found by climbing up iteratively over a segmentation of the QGM (sub-QGM's), of the "tree". The size of a sub-QGM is capped by the same predefined threshold that was used in the learning phase (identified by the number of joins). We verified, in practice, that up to four joins is optimal. This process is recursively applied until the stopping LOLEPOP denoted as `RETURN` is found in the QGM's RDF graph. After all plan transformations are applied, the recommendation guidelines are collected into a guideline document. The original query coupled with the guideline document is then passed through the optimizer again for re-optimization.

When GALO can "re-optimize" a query, it creates a guidelines document that contains the matched rewrites from the knowledge base that apply. This guidelines document is submitted with the query for optimization before execution. In this way, the query and the guidelines are passed to the query optimizer to produce a query execution plan. This is a more general, and safer, way to perform re-optimization than, say, would be adding *pragma* to the query (to represent the "rewrites"), which would force the chosen rewrites to be applied. The collection of rewrites that matched might not all apply within the plan; application of one might lead the optimizer to an altered plan in which the others no longer apply. This way, the optimizer applies the guidelines that are consistent within the course of current query planning for the query at hand.

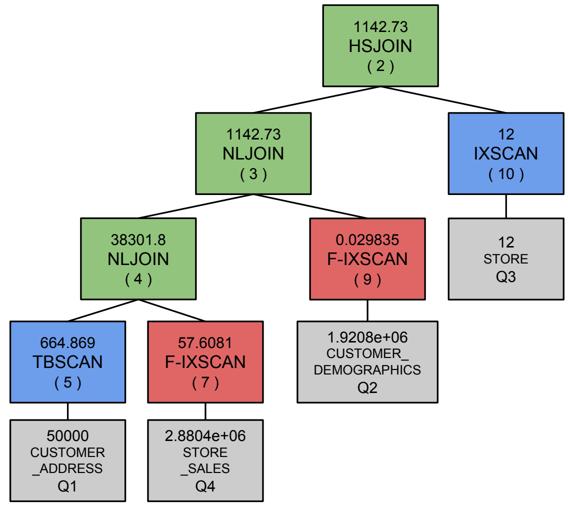
The GALO system automates and routinizes query performance plan checks by running a general test of all discovered problem patterns against a given query workload. Since problem pattern templates are abstracted with canonical symbol names and cardinality ranges, the system is not limited to a specific query workload, as problem patterns learned over one query workload can be employed in another query workload.

Consider the under-performing QGM as chosen by the optimizer illustrated in Figure 7 from a real-life IBM client query. The under-performance can be mostly attributed to a wrong estimation of the cost of the scans on tables `CUSTOMER_DEMOGRAPHICS` (`Q2`) and `STORE_SALES` (`Q4`). For these tables, the optimizer has overestimated the cost of the `TBSCAN`, and underestimated the cost of the `IXSCAN` for each table. While favoring the `IXSCAN` would be good for concurrently accessing data since less locking is involved (in environments where there is large concurrency window), compared to `TBSCAN`, we focus mainly on the actual performance of the query. The overestimation of the `TBSCAN` can be seen when analyzing the QGM file represented in Figure 7b.

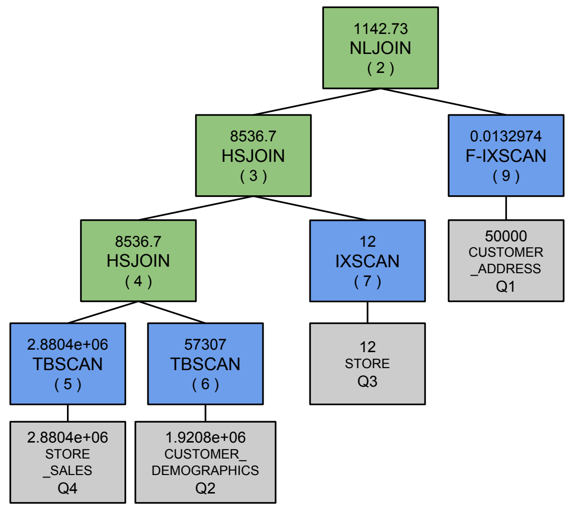
The `TBSCAN` #6 over the table `CUSTOMER_DEMOGRAPHICS` (`Q2`) has the estimate cost of 208,909, while the total cost of the whole plan in Figure 7a is 207,647. The optimizer has estimated that the whole latter plan is less expensive than the `TBSCAN` #6 from the former plan. A solution for fixing this would be to reduce the transfer rate property in the database. The transfer rate is a property that refers to the transfer rate of the disk from which the DBMS is loading the data. Reducing the aforementioned property would fix the overestimation of the replacement QGM in Figure 7b. GALO, by replacing the under-performing sub-QGM in Figure 7a with the one in Figure 7b. speeds up the query execution ten times!

An additional problem performance pattern related to sorting is presented in Figure 8a over the query from the TPC-DS benchmark. The problematic portion of the query is the join between the `STORE_SALES` fact table (`Q1`) and the `DATE_DIM` dimension table (`Q2`). The table `DATE_DIM` has a range of roughly 200 years and the query ranges over the first 100 years of sales. When selecting the best plan, the optimizer mistakenly assumes that 100 years of sales are matched between `STORE_SALES` and `DATE_DIM`, however, in practice over the instance only the last year contains sales. Thus, `HSJOIN` is chosen between `STORE_SALES` and `DATE_DIM` with the `TBSCAN` access method. The result is then joined through another `HSJOIN` with the `ITEM` table, the latter accessed via an index scan (`IXSCAN`). This execution plan suffers from the costly `HSJOIN` #3 operation. Though the table `DATE_DIM` is relatively small, when joined with the large table `STORE_SALES`, it becomes very expensive due to the full scan on the fact table and the random I/Os that follow.

A fix to this query plan found by our system is to apply `MSJOIN` between the `STORE_SALES` and `DATE_DIM` tables instead. The optimization is derived from the fact that since both inputs are sorted, as soon as no more matches are found in the inner table (`DATE_DIM`), the join operation can be safely interrupted. Our system finds this pattern as it allows to keep historical information about the estimated and actual cardinalities over operators. The scan reduction proved effective upon further analysis over the selected plans. LOLEPOP #4 had an initially estimated cardinality

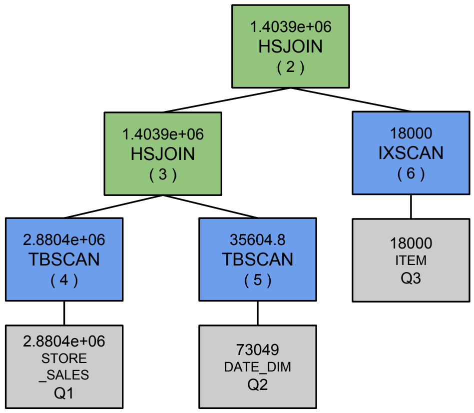


(a) Plan selected by the optimizer.

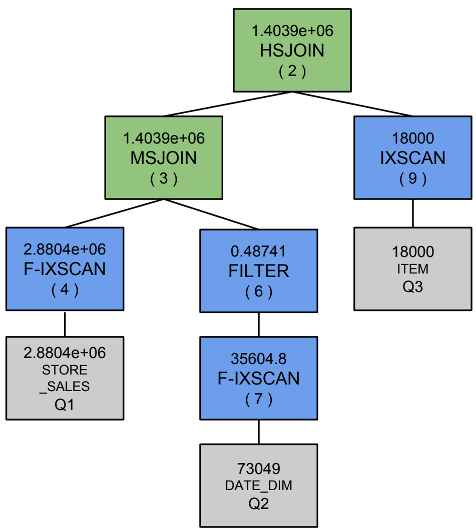


(b) Plan chosen by the GALO system.

Figure 7: Problem pattern with transfer rate.



(a) Plan obtained by the optimizer.



(b) Faster plan found by GALO.

Figure 8: Problem pattern related to sorting.

of 2.8804+e06 in Figure 8a, which was drastically reduced to the actual 550,597 rows in Figure 8b, thus, providing a near 40% overall speedup in execution time.

4. EXPERIMENTAL STUDY

We present experimental evaluation of GALO for *scalability*, *effectiveness*, and *cost* and *quality*.

1. *Scalability.* We demonstrate the scalability of GALO with respect to varying parameters of the workload and of the knowledge base that GALO builds.
2. *Effectiveness.* We report the performance gain of IBM DB2 with GALO versus without.
3. *Cost and Quality.* We compare the rewrites learned by GALO against those learned manually by IBM experts by cost of discovery and by quality of the rewrites.

We consider the ramifications of the workload *complexity* as measured by the number of LOLEPOP's in plans, and the workload *size*. We also consider the ramifications of the knowledge-base complexity, as measured by the number of tables to be joined that are permitted within the segmented sub-queries. Our experiments were conducted on servers with a 32 Intel(R) Xeon(R) CPU E5-2670 2.60GHz processor. We conducted experiments over the synthetic TPC-DS benchmark (with 99 queries), and real-world IBM client workload (with 116 queries) with database size of 1GB (and main memory adjusted accordingly to simulate real-world environment). Each query was run multiple times to eliminate noise.

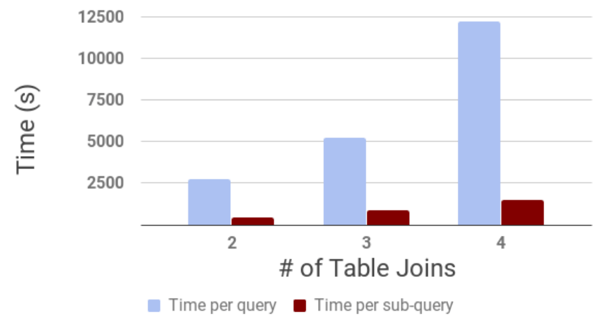


Figure 9: Learning Scalability.

4.1 Learning-Engine Evaluation

Exp-1: Learning Scalability and Effectiveness. We measure the *scalability* and *effectiveness* of the offline learning engine. To discover rewrites, SQL queries from the workload are decomposed into sub-queries, up to a predefined *join-number threshold* (number of tables to be joined). We analyze first the results over TPC-DS, and measure the average time to analyze each query (and sub-query) with varying predicate ranges. The analysis generates alternative plans via the IBM DB2 Random Plan Generator. We partition the queries to distribute to several servers to speed up the performance. Note that the sub-queries with the same structure over different queries can be merged and evaluated once.

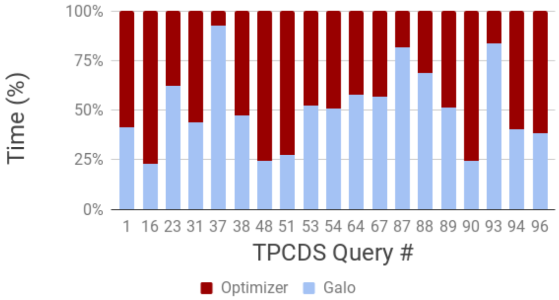
We report the results in Figure 9. The average time to analyze each query grows exponentially as the join-number is raised (as all combinations of joins must be considered), however, this is controlled by the table join-number threshold. The average time to analyze each sub-query grows linearly as the join-number is increased.

On the one hand, when the join-number threshold is too low, we do not discover pertinent problem patterns. On the other hand, when the join-number threshold is too high, there are diminished returns at great expense. Few additional problem-patterns are discovered; and these rarely match during the online plan re-optimization, due to the low probability a large structure will match. We verify that, in practice, a threshold of *four* provides the most optimal matching improvements. When this threshold is held constant, the system scales linearly with respect to workload size and complexity. Thus, the system scales well to large query workloads.

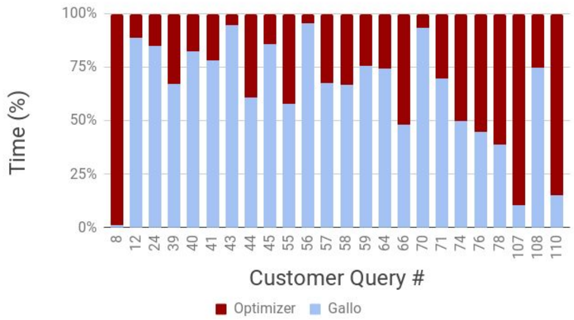
Applied to TPC-DS, the learning engine populates the knowledge base with 98 problem pattern templates. The average performance improvement of the rewrites discovered for TPC-DS is 37%. We observe similar trend over the real-world IBM client query workload with 116 queries, where additional 178 problem pattern templates are learned. The average improvement of the rewrites is 35%. The average time per query to populate the knowledge base is reasonable and practical, as the computation is done offline over the IBM systems during the non-peak hours, and in parallel over multiple machines (as described in Section 3.2).

4.2 Matching-Engine Evaluation

Exp-2: Matching Performance Improvement. We report the performance improvement of re-optimized plans via GALO accomplished by the matching engine compared



(a) TPC-DS queries.



(b) IBM client queries.

Figure 10: Optimizer with GALO versus without.

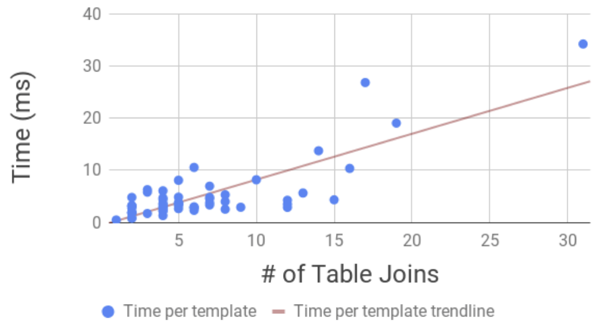


Figure 11: Matching time in # of table-joins.

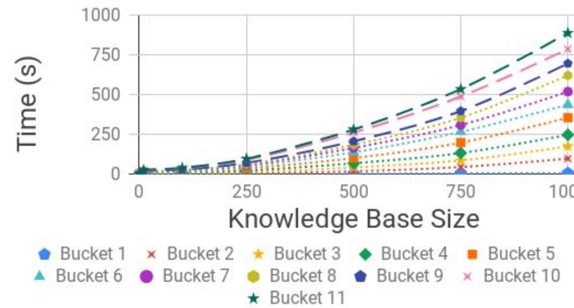


Figure 12: Matching Engine Routinization.

against the plans without re-optimization (those chosen initially by IBM DB2). We also quantify the number of problem patterns that overlap between query workloads.

The performance improvement results for the TPC-DS benchmark and the real-world IBM client query workload are presented in Figure 10a and in Figure 10b, respectively. The runtime for each query is normalized to "100%" with respect to the runtime for the original plan. Thus, the red plus blue represents the runtime for the original plan, while the blue bar represents the time for the re-optimized plan (including the time to perform the rewrite that is marginal).

The performance gains are dramatic. For the TPC-DS workload the average performance gain by the re-optimization is 49%. For a real-world IBM client workload, the average performance gain is 40%. A significant proportion of the queries were matched for re-optimization: 19 queries of the TPC-DS's 99 queries, and 24 queries of the 116 IBM client's. For instance, for query #8 from the IBM client workload GALO reduced the query runtime from nine hours to just five minutes. Performance for every one of the matched queries was improved by the query rewrite.

We also quantified the number of problem patterns learned over the TPC-DS workload (Exp–1) that matched for the re-optimization over the IBM client workload. This experiment was performed to demonstrate the re-usability of problem patterns learned over different query workloads. Interestingly, six out of 23 queries that were improved by GALO's re-optimization (26%) of the IBM client's workload were by a rewrite that had been learned under the TPC-DS workload. This validates that our system is not limited to being workload specific. A predetermined library of problem patterns collected over various query workloads, stored in the collaborative knowledge base, can be matched against a given query workload by adapting automatically the dynamic context of table and attribute names.

Exp-3: Matching Scalability. We examine the scalability of the rewrite matching against workloads of varying complexity, as measured by number of tables to join within the workload's queries. Queries in modern-day workloads are quite complex. The number of tables to be joined in the TPC-DS queries varies from one to 31.

The results are reported in Figure 11. The queries have been partitioned into buckets based on their join numbers. The average time is then reported over each bucket. Even in the case when the queries have 32 tables to join, the system is able to perform the match in 34 milliseconds per rewrite. In the less complicated case of join-number 15, it takes 4.3 milliseconds. This cost is marginal since the time to run actual queries is minutes or hours. Overall, the trend is linear in the number joins. For the real-world IBM client workload, we observed similar results.

Exp-4: Routinization. We next examine the scalability of the matching engine to the size of the workload and to the number of problem patterns in the knowledge base. We partition the workload into buckets, with the number of QGM's increased by ten each time, and the number of rewrites up to one thousand. We report the results in Figure 12 over TPC-DS. This shows that the system scales well for large workloads with many problem patterns. For example, to match the 99 TPC-DS queries against the 98 learned problem patterns in Exp-1, the average time is 41 seconds. For the queries of the real-world IBM client workload, the average time to match the 116 queries against the 178 learned problem patterns in Exp-1 is 73 seconds. GALO can process a knowledge base with a 1,000 problem patterns against a workload with 100 queries in less than 15 minutes.

4.3 Comparative Cost & Quality Study

Exp-5: Cost of Learning. We conducted a comparative study to measure the time to perform problem determination, both manually by IBM experts and automatically by GALO's learning engine. This experiment is over a sample of four problematic queries, due to the limited time IBM experts could spend to participate in the experiment (as manual determination is exceedingly time consuming.)

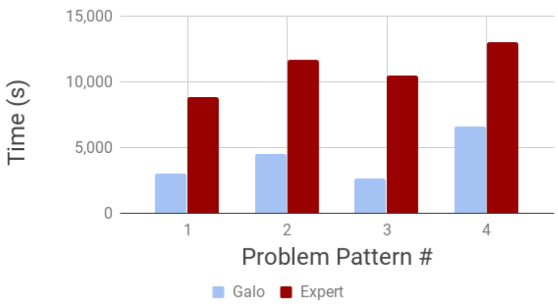


Figure 13: Time to learn problem patterns.

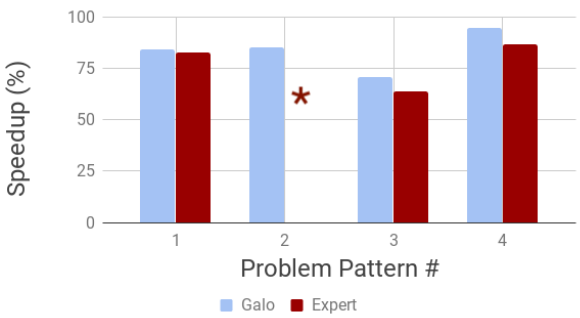


Figure 14: Quality of learned problem patterns.

We present the results in Figure 13. For the IBM experts, we report the average time, as four experts participated in the study. This experiment shows that manual problem determination is highly time consuming. On average, it is more than twice more expensive than the automatic learning by GALO, which can be computed offline. Thus, using our automatic approach, companies can save significant effort and cost as the process is fully automatic. Note that, in many cases, only the vendors' experts are skilled enough to resolve complex query performance issues.

Exp-6: Quality of Learned Problem Patterns. We also conducted a quality analysis of plans obtained manually by IBM experts and automatically by GALO. Figure 14 reports the percentage improvement of plans as found manually and automatically against the supposed optimal plans as generated "maliciously" by the DB2 optimizer.

This illustrates that the manual learning is significantly less effective than the automatic learning. For three of the problem patterns (#1, #2 and #4), IBM experts found fixes that improved the optimizer performance; however, the replacement plans they found are not as good as those found by GALO. The experts were not able to find any fixes for problem-pattern #2 (denoted with * symbol); GALO identified and resolved the issue.

For the query in Fig. 4a, the experts identified the costly join in the `NLJOIN` #2; they changed the plan to that in Fig. 15. Their new plan is faster, an 82% improvement, as it does not compute the expensive `FETCH IXSCAN` on the `CATALOG_SALES` table (`Q2`) for each row in the outer input. While their improvement is significant, the plan chosen by GALO improves over the experts' plan by another 8.6%.

We observed also during this experiment that problem determination is prone to human errors. Misinterpretation was common; for example, the value for a property in a LOLEPOP of a QGM can be easily confused, since it can appear in either decimal (e.g., 13.1688) or exponential format (e.g, 1.441e+06), as seen in Figure 15.

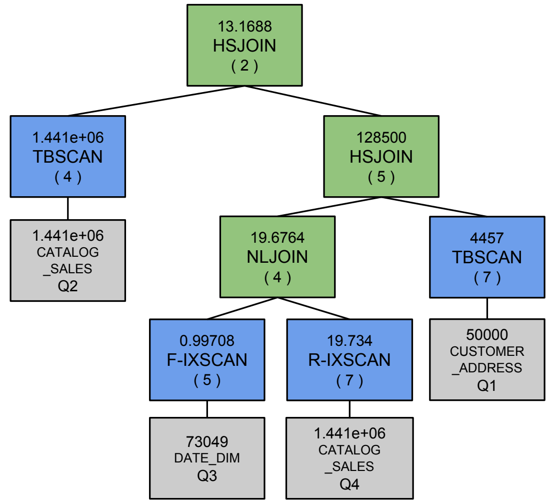


Figure 15: Expert's plan for problem in Figure 4a.

5. RELATED WORK

Query optimization has been fundamental to the success of relational database systems since their genesis. System R [20], the seminal architecture for cost-based optimization, established the importance of access-path choice, index usage, join ordering, and pipelining, and the use of cardinality estimation and predicate reduction factors for guiding plan construction in a cost-based way. Selinger's join-enumeration algoritm at the core of System R uses dynamic programming to construct a query plan bottom-up, thus "enumerating" through a vast plan space more efficiently. Over the decades since, there has been vast advances in query optimization, both in research and development. This work is driven as workloads, SQL, and applications become more complex, always moving the goal line.

Cracks in the foundations of the mainstay approaches have begun to appear, however, with big data applications and ultra-complex SQL queries [12, 22].There are two reasons for this. First, plans found by the optimizer are rarely now optimal. A System-R style optimizer is guaranteed to find the best plan, *modulo* cost-estimation accuracy and logical compromises, such as not exploring bushy trees. Estimation inaccuracy increases with complexity of queries and system configuration. Second, the dynamic-programming core of the optimizer cannot scale to very complex queries. Our work here is directed towards addressing this first crack.

Vendors have long offered automated tools for troubleshooting performance issues: IBM DB2 Design Advisor [24, 25], IBM Optim Query Workload Tuner [1], Oracle SQL Access Advisor [7], and Microsoft Database Engine Tuning Advisor [2]. While such advisors are quite useful for resolving general performance issues, they generally are not fine-grained for resolving issues at the level of plan "debugging".

Vendors also have introduced low-level tools for experts to troubleshoot performance issues for when the optimizer fails to choose optimal plans [4, 5, 7, 23]. Oracle offers *pragma* in its SQL, and Microsoft SQL Server offers *hints*, submitted with the query to override optimizer's decisions. IBM introduced *guidelines*, an XML document submitted with query to the optimizer, to redirect the optimizer's decisions. However, such manual debugging is cumbersome and time consuming, and the performance issues are often subtle. The

OptImatch system [9, 10] lets experts feed problematic query plan patterns and their resolutions into a *knowledge base*. The knowledge base is built by hand, though. GALO automatically discovers the problem patterns. (System demonstration is described in Damasio et al. [8].)

Incorrect cardinality estimation by the optimizer is a key factor leading to sub-optimal plans. In [15], a neural network is applied to improve cardinality estimation. StatAdvisor [11] is a system for recommending statistical views for the workload and improving database statistics that is crucial to cost-based optimizers. Another approach to improving cost estimation is to refine automatically the optimizer's cost model. In [6, 14], they introduce *self-tuning* cost models.

While our work addresses the first fault of sub-optimality, via re-optimization, there is wide work—albeit primarily academic—on addressing the second fault of optimization scalability. These two general efforts are orthogonal; solutions can be combined. A new generation of genetic algorithms for query optimization have been introduced, starting with [18], as an alternative to the traditional dynamic programming techniques. In [16, 19], deep learning techniques are explored for state representation and join enumeration.

6. CONCLUSIONS

We introduce a novel automatic system, GALO, that uses RDF and SPARQL to discover problematic problem patterns and provide recommended fixes. GALO offers a third stage of optimization, *plan rewrite*. This re-optimization leads to significant improvement in workload performance.

In the future work, we plan to apply machine learning techniques, such as deep learning, to learn query problem patterns by varying parameters. We also plan to develop a distributed framework to effectively partition and load balance computation to improve further GALO's performance.

7. REFERENCES

[1] IBM infosphere optim query workload tuner. https://www.ibm.com/support/knowledgecenter, 2018.

[2] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *SIGMOD*, pages 930–932, 2005.

[3] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

[4] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Interactive plan hints for query optimization. In *SIGMOD*, pages 1043–1046, 2009.

[5] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power hints for query optimization. In *ICDE*, pages 469–480, 2009.

[6] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 24(2):177–228, 1999.

[7] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB*, pages 1098–1109, 2004.

[8] G. Damasio, S. B. V. Corvinelli, P. Godfrey, P. Mierzejewski, J. Szlichta, and C. Zuzarte. GALO: Guided Automated Learning for re-Optimization. *PVLDB*, 12(12):1778–1781, 2019.

[9] G. Damasio, P. Mierzejewski, J. Szlichta, and C. Zuzarte. Optimatch: Semantic web system for query problem determination. In *ICDE*, pages 1334–1337, 2016.

[10] G. Damasio, P. Mierzejewski, J. Szlichta, and C. Zuzarte. Query performance problem determination with knowledge base in semantic web system optimatch. In *EDBT*, pages 515–526, 2016.

[11] A. El-Helw, I. F. Ilyas, and C. Zuzarte. Statadvisor: Recommending statistical views. *PVLDB*, 2(2):1306–1317, 2009.

[12] J. Gryz, Q. Wang, X. Qian, and C. Zuzarte. SQL queries with CASE expressions. In *ISMIS*, pages 351–360, 2008.

[13] R. Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In *VLDB*, pages 481–492, 2005.

[14] Z. He, B. S. Lee, and R. Snapp. Self-tuning cost modeling of user-defined functions in an object-relational dbms. *TODS*, 30(3):812–853, 2005.

[15] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality estimation using neural networks. In *CASCON*, pages 53–59, 2015.

[16] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *aiDM*, pages 3:1–3:4, 2018.

[17] B. McBride. Jena: Implementing the rdf model and syntax specification. In *ICSW*, pages 23–28, 2001.

[18] V. Muntés-Mulero, J. Aguilar-Saborit, C. Zuzarte, and J. Larriba-Pey. Cgo: A sound genetic optimizer for cyclic query graphs. In *ICCS*, pages 156–163, 2006.

[19] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM*, pages 4:1–4:4, 2018.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[21] D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, pages 57–67, 1996.

[22] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization. *PVLDB*, 10(7):721–732, 2017.

[23] M. Ziauddin, D. Das, H. Su, Y. Zhu, and K. Yagoub. Optimizer plan change management: Improved stability and performance in oracle 11g. *PVLDB*, 1(2):1346–1355, 2008.

[24] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004.

[25] D. C. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. M. Lohman, R. J. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, and G. Valentin. Recommending materialized views and indexes with the ibm db2 design advisor. In *ICAC*, pages 180–187, 2004.