GALO: Guided Automated Learning for re-Optimization

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

ABSTRACT

Query performance problem determination is usually performed manually in consultation with experts through the analysis of query plans. 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. GALO's knowledge base is built on RDF and SPARQL, which is well-suited for manipulating and querying over SQL query plans, which are graphs themselves. It then uses the knowledge base online to re-optimize queries queued for execution to improve performance, often quite dramatically.

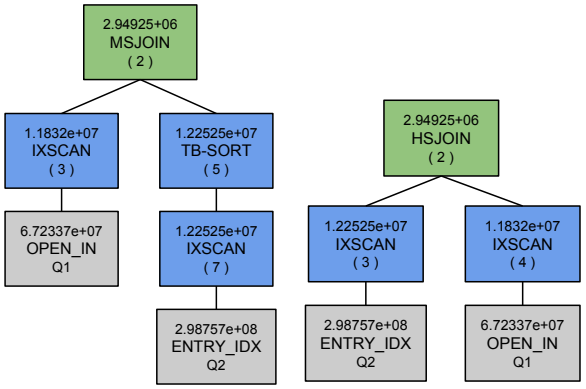
PVLDB Reference Format:
Guilherme Damasio, Spencer Bryson, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. GALO: Guided Automated Learning for re-Optimization. *PVLDB*, 12(12): 1778-1781, 2019.
DOI: https://doi.org/10.14778/3352063.3352064

1. INTRODUCTION

As the complexity of database queries and schemas spiral ever upward, the challenges facing database systems have become severe [1]. SQL queries nowadays often are generated by middleware tools instead of by SQL programmers. Business-intelligence platforms, such as IBM's Cognos have enabled organizations to scale data analysis systematically as never before. Meanwhile, the SQL queries generated "behind the scenes" can span hundreds lines of code.

Query optimization has long been a hallmark of data warehouse systems, which has enabled the data-analysis revolution. However, the complexity of modern workloads is outpacing what database systems can perform efficiently. Database optimizers fail to pick best query plans more often.

Consider the portion of the tree of the query plan chosen by IBM Db2 as "optimal" shown in Fig. 1a. 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` gets the `ENTRY` table data using an index scan `IXSCAN` (#7) of the index `ENTRY_IDX` that contains all the column values needed, and then performs a sort that is read by a table scan, `TB-SORT` (#5). The size of data entering the sort and the number of pages that spill to the disk at runtime are large.



(a) Plan by IBM Db2. (b) Plan by GALO.
Figure 1: IBM client query with a problematic join.

The chosen *fix* by our GALO system[1] shown in Fig. 1b has exchanged the `MSJOIN` with a *hash join* (`HSJOIN`), and swaps the *outer* and *inner* tables. While the `HSJOIN` spills pages into the disk at runtime too, the amount is significantly smaller. This plan rewrite reduced the query runtime from 9 hours to just 5 minutes! IBM experts report that even more complex patterns exist, which can take days to be resolved.

Database vendors have made raw tools available to SQL programmers to troubleshoot performance problems. Oracle and Microsoft offer *pragma* and *hints*, respectively, embedded in their SQL, while IBM provides *guideline* documents (written in XML) submitted with a query to the optimizer. Pragma, hints, and guidelines serve to sway the optimizer's choices in query planning. They can be used by the programmer, by profiling the query plans and execution traces, to override decisions that the optimizer would make concerning, for example, choice of join algorithms and join order.

Such manual performance debugging, however, has become increasingly difficult with very complex query workloads. More time than ever is now spent by experts on such tasks at the major database-vendor companies. Thus, automated tools are needed for the workload debugging. Existing tools do not have the means of doing a deeper parse of

[1]System video available at https://vimeo.com/335947845

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352064

query plans that would allow for automated reasoning and querying over problem patterns that GALO does.

This workload debugging also has been *ad hoc*. The lessons learned from one fix are lost, to be rediscovered by others later. At IBM, our recently developed OptImatch system [3, 4] has been a successful effort towards addressing this. Experts feed problematic query-plan patterns and their resolutions into an OptImatch *knowledge base* to be shared with others. Still, the OptImatch knowledge base is built by experts tediously by hand. GALO significantly extends over OptImatch as the knowledge base is *automatically* "learned". We use the RDF graph model and the SPARQL language to represent and query execution plans.

Today's database optimizers are two stage: a *query rewrite optimizer* and a *cost-based optimizer* [1]. There might be "flaws" in the chosen query plan, however. Cost estimations may go awry. Unusual characteristics in the data and the query can circumvent the planning strategies as encoded in the database optimizer. GALO offers a *third tier* of *re-optimization*. Rules from GALO's knowledge base are applied to the resulting query plan that resolve known performance trouble spots.

GALO can be also used by the performance optimization team to extract from the knowledge base those systemic issues, to learn and develop new rewrite rules for query rewrite and optimization techniques 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.

The GALO system builds on our foundations developed in [2]. In the demonstration, we showcase a novel system for problem determination that exploits graph databases. Attendees will use GALO over the synthetic TPC-DS benchmark and real IBM client query workloads.

GALO's *knowledge base* is 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.

The *learning engine* 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 query's predicates are varied to result in different cardinalities. Then, for each sub-query from this, 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 better than the optimizer's, it flags the pair as a rewrite, a problematic plan pattern and its guideline solution.

The *matching engine* 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 learned over a different query, even from a different query workload.

2. THE GALO SYSTEM

System Overview. GALO is an automated system to improve SQL workload performance. A *workload* is a popu-

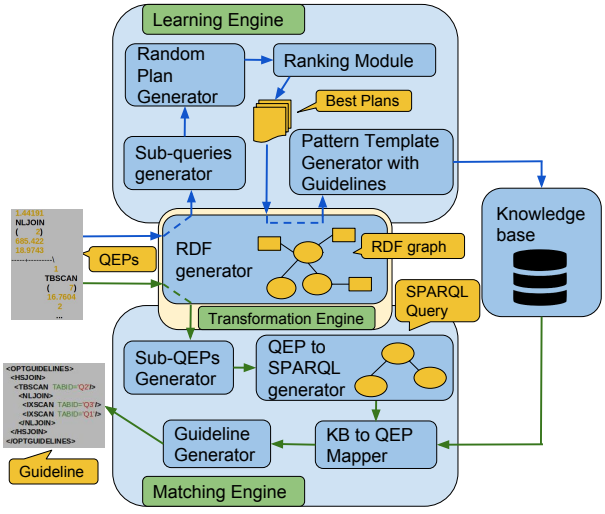


Figure 2: System architecture.

lated database with a requisite schema and a collection of SQL queries that are periodically executed. GALO *profiles* the workloads *offline* to construct a *knowledge base* which captures performance issues 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 manual OptImatch system [3, 4]. As the "second generation" of OptImatch, GALO is used within IBM to populate automatically a general knowledge base that tracks query plan issues. GALO is also used as a resource for the IBM Db2 team for evolving and refining the Db2 optimizer based on rewrites in the knowledge base.

GALO's system architecture is illustrated in Fig. 2. The system is comprised of a *transformation engine*, a *learning engine*, and 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 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.

Transformation Engine. A *query execution plan* (QEP) is the plan constructed by the query optimizer for an SQL query. Within IBM Db2, query specifications are represented in the *query graph model* (QGM). An SQL query is parsed into a QGM representation. That QGM is then rewritten by Db2's query rewrite engine, which applies transformations known to simplify the query. The resulting QGM is then passed to Db2's cost-based optimizer, which goes through a process of table access and join enumeration to come up with a full fledged query execution plan. Fig. 1 depicts a portion of the query plan file. 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

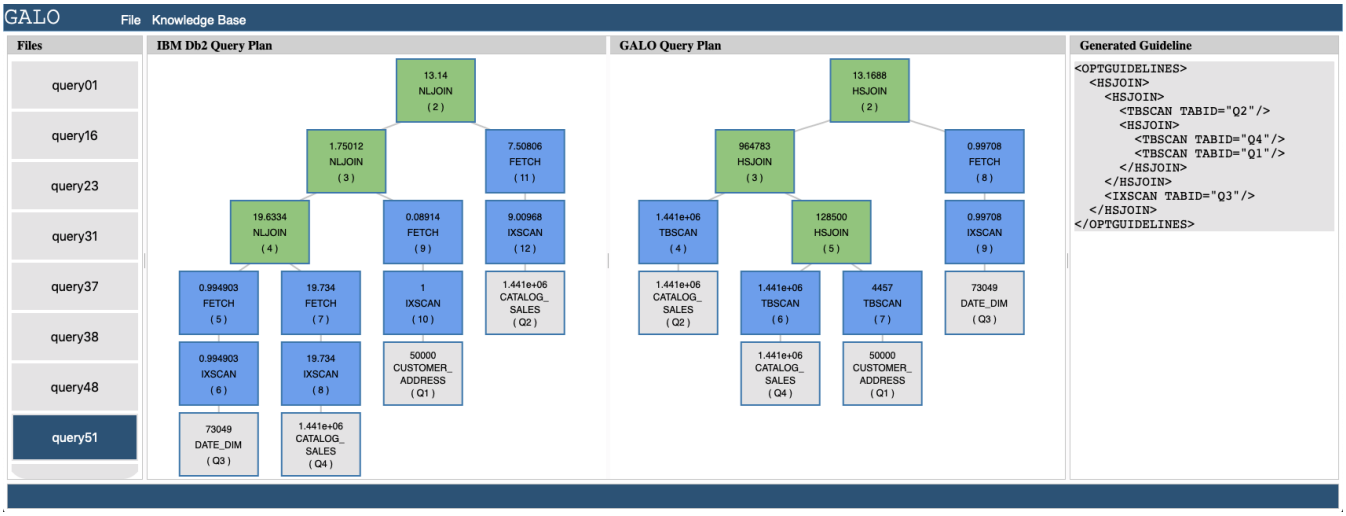


Figure 3: GALO GUI with hash-join bloom filter problem pattern.

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.

The transformation engine is GALO's tool to map QGM's and SQL queries into RDF graphs. 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 resulting RDF graph is a full transformation of the text-based QGM. GALO uses the Apache Jena RDF API to map the QGM into an RDF graph.

Learning Engine. The learning engine populates the knowledge base with problem pattern templates and their counterpart recommendations. 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. 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.

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 joins and applicable local predicates over selected tables from the original query. 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 patterns with the same best plan within lower and upper-bound cardinalities. This ensures 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. 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.

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" within the predicate property ranges.

Detected query problem patterns are transformed into *templates* to be saved in the knowledge base RDF graph. This is a critical abstraction step, which 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 substructure and characteristics, but with different table and attribute names, are matched against the same problem pattern templates. This assures that problem pattern usability is not limited to a specific query or query workload.

The recommended replacement patterns for corresponding "malicious" problem pattern templates are stored in the knowledge base as XML *guidelines* (Fig. 3). A problematic portion of QGM as chosen by the optimizer from the TPC-DS workload query is shown in Fig. 3. At runtime, the `F-IXSCAN` (#7) suffers from excessive random I/O reads. This is a consequence of a poorly clustered index used to access the `CATALOG_SALES` table, causing pages to be loaded into the buffer pool as usual, but then being overwritten by other pages subsequently loaded. This adds significantly to the I/O. 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. This overhead is propagated upward into the next `NLJOIN` (#2) and operators to follow, causing further performance issues.

The discovered solution by GALO in Fig. 3 applies a hash-join bloom filter in the `HSJOIN` (#2). A bloom filter is a space-efficient, probabilistic data structure to test if an element is a member of a set by hashing the values. In the better query plan, the hash join creates a bitmap from the

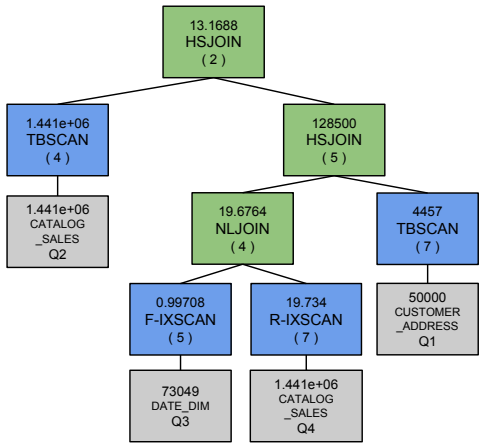


Figure 4: Expert's plan for problem in Figure 3.

inner input. This is used as a lookup for the join to avoid hash-table probes for outer tuples that never can match. This results in an approximately twice faster query plan.

Matching Engine. At runtime, a large SQL query to be re-optimized is segmented into sub-queries. 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.

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.

When GALO can "re-optimize" a query, it creates a guideline document that contains the matched rewrites for the query plan from the knowledge base that apply. This guideline document is submitted with the query for re-optimization before execution to produce a query plan. This is a more general and safer way to perform re-optimization than to "patch" the plan by forcing all the chosen rewrites to be applied, which could result in incompatibilities in the overall plan. The 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.

3. DEMONSTRATION PLAN

We design scenarios for three demonstration objectives illustrating our web-based system: a) *Familiarization* b) *Challenges* and c) *Effectiveness*. We deploy the synthetic TPC-DS workload (with 99 queries) and the real IBM client workload (with 116 queries). Due to the IBM copyright restrictions, GALO is not available online to general public; thus, the audience will use our machines to gain the access.

To *familiarize* users with our methodology, we first demonstrate GALO's functionality over a number of queries with simple problems. We select queries, such that the whole *learning* and *matching* process is comprehensible when visualized step-by-step.

To highlight the *challenges* of the complexity of modern workloads outpacing database systems, we carefully choose a number of non-trivial large and complex queries. We present the audience with the query at hand, and how to design an efficient executable query plan using the capabilities offered by GALO employed *online* to re-optimize the query plans of incoming queries with the knowledge base for plan rewrites. We focus on the interesting challenges that promote query performance, including efficient join order and type selection, cost and cardinality estimation, random I/O reads, indexing and sorting. We also show the participants that problem patterns overlap between query workloads. For instance, six out of 24 queries that were improved by GALO's re-optimization (25%) of the IBM client's workload were by a rewrite that was learned under the TPC-DS workload.

To demonstrate *effectiveness* of our system, we compare the rewrites learned by GALO against those learned manually by the participants by cost of discovery and by quality of rewrites. To simulate a real-world environment, the participants are allowed to access the tools that experts use in their daily problem determination tasks. This includes *grep* command-line utility for searching plain-text query plans for lines matching a regular expression. We plan to observe the participants during the demonstration—as we have observed IBM experts using GALO in use case studies—to see the types of errors they make. Misinterpretation is common; e.g., the value for a property in a LOLEPOP of a QGM has been confused by IBM experts, as it can appear in either decimal (e.g., 13.1688) or exponential format (e.g, 1.441e+06), as seen in Fig. 4.

As an example, we measured the time and conducted quality analysis to perform problem determination both manually by IBM experts and automatically by GALO learning engine over a sample of four queries. Manual problem determination is highly time consuming on average three times more expensive than the automatic learning by GALO. For three of the four problem patterns, IBM experts found fixes that improved the optimizer performance, however, the replacement plans they found are not as good as those found by GALO. GALO identified and resolved all the issues with 80% average performance speed-up over IBM Db2.

For the query in Fig. 3, the IBM experts identified the costly join in the `NLJOIN` #2; they changed the query plan to that in Fig. 4. Their new query plan is faster by 82% than the IBM Db2 system, as it does not compute the expensive `FETCH IXSCAN` on the `CATALOG_SALES` table for each row in the outer input. The plan chosen by GALO improves over the experts' plan by another 8.6% to a total of 90.6%.

4. REFERENCES

[1] 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.

[2] G. Damasio, V. Corvinelli, P. Godfrey, P. Mierzejewski, A. Mihaylov, J. Szlichta, and C. Zuzarte. Guided automated learning for query workload re-optimization. *PVLDB*, 12(12):2010–2021, 2019.

[3] G. Damasio, P. Mierzejewski, J. Szlichta, and C. Zuzarte. OptImatch: Semantic web system for query problem determination. In *ICDE,* pp. 1334-1337, 2016.

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