DistGALO

Alexandar Mihaylov
Ontario Tech University, Canada
Oshawa, Ontario
alexandar.mihaylov@uoit.ca

Spencer Bryson
Ontario Tech University, Canada
Oshawa, Ontario
spencer.bryson@uoit.ca

Vincent Corvinelli
IBM Centre for Advanced Studies, Canada
Unionville, Ontario
vcorvine@ca.ibm.com

Parke Godfrey
York University, Canada
Toronto, Ontario
godfrey@yorku.ca

Piotr Mierzejewski
IBM Centre for Advanced Studies, Canada
Unionville, Ontario
piotrm@ca.ibm.com

Jaroslaw Szlichta
Ontario Tech University, Canada
Oshawa, Ontario
jarek@uoit.ca

Calisto Zuzarte
IBM Centre for Advanced Studies, Canada
Unionville, Ontario
calisto@ca.ibm.com

ABSTRACT

Query optimization is a quintessential element of modern Database Management Systems(DBMSs). Compile-time driven estimates and heuristics aid the compiler in selecting what is deemed the lowest cost Access Plan for a given query. These access plans are seldom optimal, and can oftentimes lead to under-performing query runtimes, with varying severity. To counter this, experts with years of domain knowledge must painstakingly examine the access plans in effort to determine where further improvements can be made. This approach proves to be very time consuming, prone to error and requires years of domain expertise. In previous work, we devised a fully automated solution capable of learning query problem patterns and applying them to poorly performing queries. In this work, we introduce DistGALO, a modern-day solution to meet the modern-day demand for scalable systems. With a newly overhauled Learning Engine, the system utilizes clustering and various partitioning strategies to ensure minimal skew during its learning process. We also introduce several pruning strategies, including the RSACE module which gives user fine-grained control for trading off runtime versus template creation. The newly distributed system still maintains the same objective as the previous, acting as the third access plan rewrite layer in the query optimization process. We demonstrate the efficiency boost over our previous system using the synthetic TPC-DS benchmark, and the effectiveness of our pruning strategies.

1 INTRODUCTION

1.1 Motivation

The volume of data being stored and processed is typically domain dependant, but recently there's been an immeasurable increase across all domains [34][5]. This trend can be attributed to various factors and technologies, including the widespread availability of IaaS (Infrastructure as a Service), Internet Of Things (IoT), smart devices, and consumer demand for global connectivity. The increase in data posed many challenges for database management systems and their vendors. One consequence was that SQL queries became more complex since middleware tools became capable of automatically generating them [17]. The convenience these tools brought from a user perspective, they took away from a problem determination one. This increase in complexity has led to queries with essentially no limit on the number of algebraic operators they contained, potentially spanning hundreds of lines. This though indifferent to database users has proven increasingly more difficult for domain experts conducting troubleshooting over the queries.

From an automation standpoint, database query optimizers absolve some of the manual labour experts would otherwise have to incur. The query optimization process has been researched thoroughly[20][32] and new techniques are continually being developed[30]. The underlying process relies heavily on cardinality estimates to determine a cost model for varying access plans that express the path of execution. A varying set of access plans are considered, with alternate join orders and operators, with the lowest cost plan ultimately being selected for execution. The selected and optimal access plan would ideally be identical but is not always the case since there is a large dependence on estimates and timely expectancy. When cardinality estimations prove inaccurate, the optimizer tends to pick sub-optimal access plans, and under-performance may arise.

In effort to aid database administrators (DBAs), several tools have been developed[8][7] by providing suggestions to the optimizer as to what decisions(operators) are to be chosen in the final access plan. These are denoted as *pragma* in the Oracle Database, and *hints* by Microsoft SQL Server. These suggestions can be embedded into the SQL but may become dated over time as the database data and subsequently, statistics change. Db2 offers an XML based *guideline document* that provides recommendations to the optimizer during the cost-based optimization stage. This document can be submitted alongside the query and provide seamless alteration from a user perspective.

It is at this stage that domain experts must intervene to address the under-performance, by manually analyzing and adjusting the access plan through hints, pragmas, or guidelines, depending on the database vendor. This process, however, is quite laborious and requires a high degree of domain expertise. More so, with the

increasing complexity of automatically generated SQL queries, the manual process is starting to prove unbefitting. The manual problem determination is also done *ad hoc*, and thus recurring patterns are forgotten and redundant work must be performed.

Domain experts are scarce resources and can seldom devote time to the arduous troubleshooting. A more automated approach is demanded, but recent approaches to do so failed to adjust a miss-performing access plan[4][9]. We have aimed to provide a software solution, with our first semi-automated approach OptImatch [12][13]. This approach allowed domain experts to graphically create problem patterns, to later save in a *knowledge base*, and share with other domain experts. The system proved to aid the troubleshooting, but still required some manual input from experts in order to populate the knowledge base. In effort to fully automate the system, we later developed GALO[11][10]. This was a fully automated approach, which functioned as a two-step process. First the *Learning Phase* was responsible for learning problem patterns offline, and populating a knowledge base. The *Matching Phase* would then query the knowledge base online and automatically apply access plan repair on problem workloads. The system was well received, and in this work we extend it to a distributed environment, providing a modern-day solution to the growing demand for modern-day scalable systems.

1.2 Goals

The goals for DistGALO subsume the goals of the previous GALO[11] and OptImatch[12] systems, but also extend them as follows:

(1) *automatic query problem determination*;
(2) *plan re-optimization*;
(3) *optimization evolution*; and
(4) *scalability*.

DistGALO's Goal 1 is in direct response to the manual effort optimizer experts exert in finding problems within large query access plans. To address this, OptImatch was first devised so as to provide some relief in the process by allowing experts to create, save, and apply recurring access plan problem patterns from a shared knowledge base. We further enhanced this process with GALO, a system capable of fully automating the knowledge base population stage, thus eliminating any need for manual expert intervention. The knowledge base of RDF graph templates and later applied to ill-performing SQL queries in order to improve performance. RDF provides a natural mapping to graph-based representations since execution plans are also graphs, and the RDF representation could be queried using the SPARQL language.

Queries must undergo several steps before an access plan is executed, and results are returned to the user. One of these includes the query re-write, where the graphical representation of the query is mapped to a semantically equivalent, but more efficient graph [33]. The graph-based representation is then passed onto the optimizer, where various *Query Execution Plan(QEP)s* are generated, each with an associated estimated cost. Ultimately the QEP with the lowest cost is selected and goes on to be executed by the database manager. QEP costs are estimates calculated at compile-time, and may not always adequately represent the effectiveness of the access plan. Inhere lies the problem, and is the main motive behind GALO's Goal 1. Sub-optimal access plans are typically selected due to the misguided estimated costs, or because the optimal QEP was never considered from the large search space of plans. GALO was able to address this issue by analyzing a large number of query fragments, or sub-queries, to create rules of problem patterns then used to populate its knowledge base. It did so quite successfully, but it came with some inherent shortcomings. The search space of sub-queries generated proved to be quite large, and with a lack of pruning rules, sequentially processing each proved to be a time-consuming process. This deficiency we address in our current work with DistGALO, by distributing the most costly and expensive stage of the process. Also, several pruning rules allow for an even greater speedup by reducing the search space. As a whole, the goal remains the same, but the methodology to achieve it has been greatly improved upon.

We have thus far explored two major steps in the compilation process, mainly the query rewrite, and the cost-based optimization stages. Goal 2 provides a third-tier absent from the compiler that aims to rewrite the final QEP chosen by the optimizer. This mends cases where the chosen QEP by the optimizer is sub-optimal, and a better performing QEP′ exists, or was undiscovered during the plan generation stage. Stored rules within the knowledge base can be queried and applied to transform an under-performing QEP, to a QEP′ that is unhindered by poorly selected operators that experts might otherwise toil over. All the acquired wisdom is aggregated within a single knowledge base, rather than throughout numerous experts, who can only rely on their ad hoc observations to trace the problematic section.

Upon locating a problematic segment, one approach would be to apply the matched rewrites directly to the QEP supplied by the optimizer, but this could ultimately result in incompatibilities in the overall QEP. Instead, DistGALO generates a guideline document that can be directly appended to the original query, without any user intervention. The query can then be passed through the compiler's pipeline, including the query rewrite and cost-based optimization stages, ensuring that all intermediate LOLEPOPs are re-evaluated and correctly estimated. The optimizer may then chose to honor the guideline, or in some cases discard it due to incompatibility issues. Ultimately if the guideline proves to be applicable, a potentially never-before-seen QEP′ may be selected, thus providing a faster execution time. This process we term *plan re-optimization*.

Goal 3 persists through GALO and DistGALO as they share many fundamental values. Both provide a valuable supply of problem patterns that can be analyzed by the performance optimization team. This analysis could potentially lead to uncovering of unknown issues lying within the optimizer, enhancing the query rewrite rules, or even applying new heuristics in selecting a more cost-optimal QEP. These improvements are not constrained to academic and benchmark synthetic data, but also can be applied to real-world customer workloads.

Goal 4 is unique to DistGALO, and was the primary motive to bring the system up to par with the vastly growing complexity of queries, and the size of data. We observed a limitation in GALO's ability to conform to more complex queries and larger datasets. This, though to some degree can be remedied by vertical scaling of the server, was not an acceptable solution. Figure 1 provides insight into the several stages involved during the GALO's learning process, and also highlights which posed as a bottleneck. The two most notable mentions, and consequently the focus of DistGALO's distribution are the *Sub-query generation* and *Random Sub-query execution* stages. The latter proved the most time-consuming taking nearly 57% of the learning process computation in GALO, with the former taking 37%. What DistGALO offers instead is a way to horizontally scale in response to the workload complexity and size. Given the maturity of the

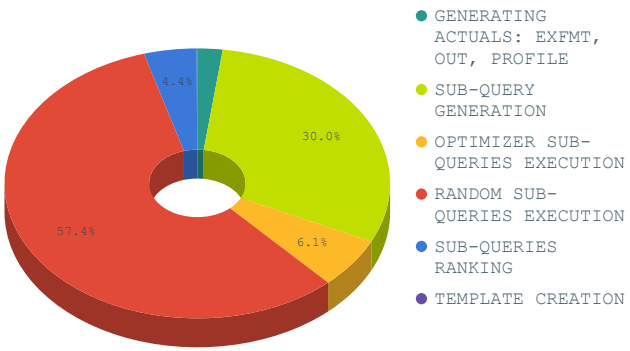


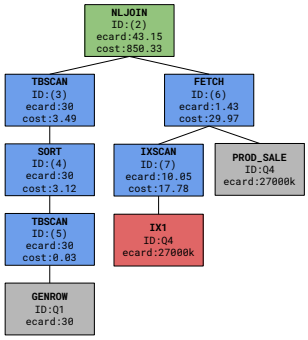
Figure 1: GALO learning process distribution.

modern-day cloud systems and their abundant availability, it is not unreasonable to rent nodes on a per need basis. If a larger workload is required, a business or even an individual is quite reasonably able to acquire more hardware ad hoc. This task is not only achievable in modern development environments, but can even be automated[1][2]. This can allow for a more optimized utilization of resources since they are now also able to scale in accordance with the work required. DistGALO aims to adhere to this growing trend by allowing horizontal scaling in response to the query and workload supplied. It utilizes a cluster of nodes that can very easily be altered, and improve performance if the use case calls for a faster response time. Currently, the system is seamlessly able to scale with a larger cluster, but the adjustment of the cluster is not done automatically just yet.

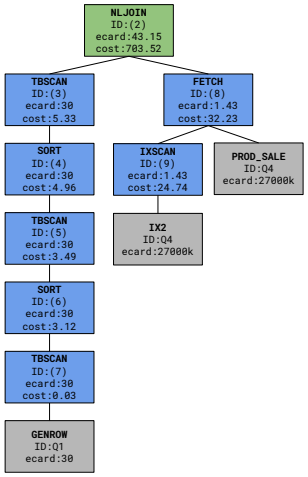
Two varying degrees of complexity cause a drastic increase in search space for GALO. First, the complexity of the queries being learned causes a larger number of table combinations, consequently leading to a larger number of sub-queries that are generated. This results in more sub-queries that would require analysis and execution, thus causing a slowdown. The other scenario is one in which the referenced tables are larger, once again causing a slowdown in the execution and probing stages of the system. DistGALO tackles both of these by incorporating multiple machines working harmoniously together, each intelligently delegated its own workload. The individual workloads ensure an overall balance and skew reduction in the system that ultimately fractions the learning runtime. Another objective was to ensure that DistGALO responded well to vertical scaling as well. Pruning rules are presented to drastically reduce the search space, whilst still providing meaningful results. The ability to scale according to the demands of the user or the workload makes DistGALO a very applicable system to utilize in real-world applications.

1.3 Real World Example

Consider a customer who experiences a slowdown in their daily system metric reports as a consequence of a slow query bottlenecking the whole pipeline. What database vendors, and subsequently domain experts, first have to do is to analyze a problematic query execution plan, like the one in Figure 2a. This already arduous task is made even more challenging when considering that the QEP shown is only a subgraph from a much larger access plan (not shown for brevity and to highlight the problem section). Even the act of identifying the subgraph is a challenging and time-consuming process that must be done manually. Each node



(a) Plan selected by the optimizer.



(b) Plan chosen by the DistGALO system.

Figure 2: IBM customer problem query and applied fix.

in the subgraph can be either a table, index, table functions, or operators such as join methods and scans. Each node we refer to as a low level plan operator (LOLEPOP) hereafter. Each LOLEPOP contains several pieces of information, but for simplicity have only shown a subset of the most relevant ones. Starting from the top, the operator type (ex. NLJOIN, TBSCAN), followed by the ID of the LOLEPOP, unique to each node, typically used for reference. Third down is the optimizer calculated estimated cardinality or *ecard* for short. Lastly, at the very bottom of the LOLEPOP is the optimizer calculated *cost* in timeron units, the estimated total I/O and CPU cost the Db2 manager might incur during execution. Values for *ecard* and *cost* might include a *k* multiplier of 1000 for brevity. Note the leaf nodes are either base tables or indexes and do not follow the formatting scheme outlined above. In leaf LOLEPOPS, rather than the operator name, we denote the name of the table or index being read from. This is followed by the instance name, the table/index referenced by the compiler post-query-rewrite. Finally, at the bottom we include the estimated cardinality for that table or index.

As a concrete example, consider the **NLJOIN** LOLEPOP in Figure 2a, which has an ID of 2, estimated cardinality of 43.15 and cost of 850.33 timerons. Next consider the base table **PROD_SALE** at the bottom-right-most. We uniquely identify it by its instance name *Q4*, which also has a cardinality of 27000*k* or 27 million. Henceforth we will refer to specific LOLEPOPs either by their unique ID or table instance name, depending on the type.

The underlying under-performance of the optimizer selected plan in Figure 2a is mainly attributed to the choice of index. The **IXSCAN (#7)** scans a large index *IX1*, using the RIDs (row ids) for the following **FETCH (#6)** above. The fix for the under-performance is a subtle one, attained by swapping the index used in the **IXSCAN (#7)** for a larger index *IX2* (with one more level in the B-Tree). The larger index (*Q4* in Figure 2b) is typically more expensive since it needs one extra I/O with each probe in the index. Since that is the only variant in the plan, the optimizer opted for the cheaper index *IX1*. The subtlety, however, lies in the values of the additional attributes present in *IX2* but not *IX1*. These post analysis, turn out to have a large number of distinct values, and predicates on these attributes qualify fewer rows that need to be fetched. Both indexes exist during complication, but the optimizer ultimately selected the cheaper inner (#6 in Figure 2a). This minute detail leads to an order of magnitude difference in runtime performance, and thus stresses the importance and need for domain experts and automated systems like DistGALO to improve the performance.

1.4 Contributions

Scalability is a critical implicit requirement for modern-day systems and remained the focus when developing DistGALO. The new system improves upon the previous GALO system, whilst still upholding the requirement of providing a third *plan re-optimization* stage. We performed a detailed analysis on a per-module basis on GALO and uncovered some bottlenecks. In this work, we have improved upon those bottlenecks by improving or overhauling components of the previous system. DistGALO aims to comply with the demand for scalable systems capable of withstanding the progressively more complex queries and larger databases. We present our contributions as follows:

(1) **Distributed Learning Component (DLC)**. First is the *Distributed Learning Component (DLC)* that transforms GALO's Learning Engine, into the Distributed Learning Engine presented in this work. Solely it is a cohesive system, but can be summarized with three of its largest components: Distributed Sub-query Generator, Query Partitioner, and the Distributed Sub-query Executor.
   (a) The *Distributed Sub-query Generator* decomposes workload queries into smaller components or sub-queries. The size of the sub-query depends on the number of joins, and thus a range can be established in the configuration of the system. Each input workload query is individually decomposed into sub-queries that are later pooled into groups. Two categories of sub-queries remain as a byproduct of the sub-query generation: the optimizer and random sub-queries. These are handled separately down the pipeline due to their nature and relationship. The distributed component assigns a workload query to a node. This can be done blindly or naively, but could potentially result in skew and therefore long execution times since all nodes must finish before proceeding. The partitioners are responsible for intelligently assigning each query to a node based on some characteristics or metrics of that query.
   (b) The *Query Partitioner* component, is used to mitigate skew throughout the cluster using various partitioning strategies. This is a critical component of the distributed environment since it determines how data should be grouped on nodes. Poorly distributing queries or sub-queries could result in nodes being overloaded and ultimately negating any inherent benefits from distributing the work in the first place. Therefore intelligently partitioning is vital in unlocking the potential of a distributed system. The partitioners rely on extracting characteristics and possibly deriving some metric from the workload. For this reason, the strategies are typically unique to the context of what is being distributed. The *Distributed Sub-query Generator* utilizes four partitioning strategies: hash, cost, join and weighted. The *Distributed Sub-query Executor* employs different strategies based on the type of sub-query being executed. For optimizer sub-queries, it opts for hash, and cost partitioning strategies. For random sub-queries, it opts for hash, a cost hybrid variant, runtime, and exchange partitioning strategies. The details of each will be elaborated upon in Section 4.3. The choice of partitioning strategies has some flexibility, and where present, can be adjusted through the system configuration file.
   (c) The *Distributed Sub-query Executor* is responsible for taking in already-partitioned sub-queries and processing them on all available nodes. Sub-queries are executed a predetermined amount of times (in effort to reduce noise), metadata from each run is collected and the best performing run is saved into the Hadoop Distributed File System(HDFS) for later processing. This component processes both the optimizer and random sub-queries separately. Optimizer sub-query runtime information is saved in a lookup table that is later used to aid the partitioner in grouping random sub-queries, and to also provide valuable threshold information during the execution stage.

(2) **Random Sub-query Abridged Clustering Executor**. Second is the *Random Sub-query Abridged Clustering Executor* (RSACE) which adds a pruning layer to minimize the processing of redundant sub-queries. This module only applies to the random sub-queries since they pose the biggest bottleneck, and also because it relies on some meta-information from the optimizer sub-query execution runs. Clusters of alike sub-queries are created, whereby only representatives from each are executed. Further execution by the *Distributed Sub-query Executor* follows this process in effort to eliminate any false positive representatives. Probe query caching and early run threshold termination are additional pruning strategies utilized by DistGALO.

(3) **Experiments**. Finally, we present *experiments* to validate the effectiveness of DistGALO, and its efficiency over its predecessor system, GALO. The effectiveness is measured over the TPC-DS decision support benchmark with synthetically generated data. We demonstrate that DistGALO is just as effective, but is also able to perform the learning process in much less time. We also quantify the performance of the varying partitioners and the RSACE module.

In Section 2 we describe some of the preliminaries required for this work. In Section 3 we provide a high level overview of the DistGALO system. In Section 4 we detail over the distributed learning process and the varying components involved. In Section 5 we validate experimentally the effectiveness and efficiency of DistGALO, followed by related works in Section 6, finally concluding our work in Section 7.

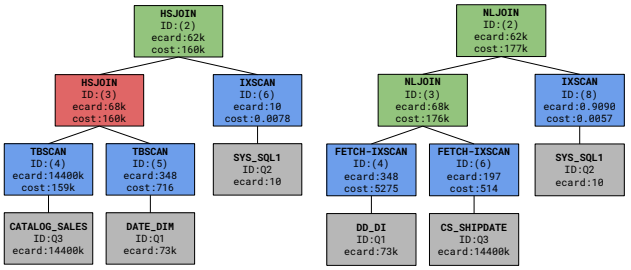
2 BACKGROUND

2.1 Preliminaries

GALO. There is an inherent problem in which database experts were exerting too much effort and time into finding fixes for problematic customer queries. To aid the laborious task of examining queries to only discover a previously seen problem pattern, OptImatch[12] was developed. It allowed experts to interact with a web interface and input recurring problem patterns they would come across. This proved an invaluable tool but still required some expert intervention. In an effort to fully automate the process, GALO[11] was developed. It was a complex system primarily comprised of a *transformation engine*, *learning engine* and *matching engine* that would all interact with a Knowledge Base(KB). The KB would be the central hub of the system, populated by the *learning engine*, and queried by the *matching engine*, using the *transformation engine* as an intermediary. The *learning engine* would profile workloads *offline*, and capture problem patterns that it would then save in a graph-based representation within the KB. A workload refers to a set of SQL queries that require some periodic execution. The *matching engine* processes a separate workload *online* whilst querying the KB for potential plan rewrites overseen by the optimizer. This enables the optimizer to use GALO as the *third tier of optimization*, be re-writing access plans using templates from its previously populated KB. The *transformation engine* acts as a translator between the language the databases uses, and the language of the knowledge base. The decision tree the optimizer selects as the path of execution it must take during runtime is represented as a graph. The KB is also built using a graph based-representation, but using a different format and thus needs to be translated by the *transformation engine*.

Query Compiler. The runtime of queries is a quintessential in the successful operation of some applications. In some cases, hundreds or even thousands of queries may be requested within a very small time period, and so retrieving the results promptly becomes even more critical. The query compiler optimizer aims to aid this process through the use of heuristics and statistics. The query must pass through several crucial steps before the optimal access plan is found and executed. Under the hood, the query is represented by a Query Graph Model (QGM) as an in-memory database. The query must first be checked to be syntactically and semantically sound, to ensure that the user did not make an error. The query is then transformed into an easier to optimize format in the query rewrite stage. Tables can be renamed here, and predicates can be pushed down to other levels of execution to improve performance. The QGM is updated and, with the help of various statistics from tables, indexes, columns, and functions, it generates various execution plans, each with an associated estimated cost. Ultimately the execution plan with the lowest cost is selected and used during runtime. As a final step, the compiler creates the executable access plan for the query ensuring it is devoid of any redundant computations.

Db2 provides an explain facility to shed some light on some of the steps outlined above and provide greater insight into decisions to be made at runtime. The explain facility can provide detailed information on what tables and indexes were accessed, cost information, statistics for all referenced objects, as well as predicates and selectivity estimates for each, to name a few. Also, it captures the sequence of operations the optimizer selected to process the query. This sequence of operations can also be represented as a graph and is termed the Query Execution Plan (QEP).



(a) Plan selected by the optimizer. **(b) Plan chosen by the DistGALO system.**

Figure 3: Problem pattern with expensive hash join and DistGALO discovered fix.

This is, in essence, a data-flow graph of operators, where edges are the flow of the data itself, and the nodes are operations like joins or sorts[33]. Note that the QEP is available at compile-time, and as such only presents estimated cardinalities and timerons, in contrast to the actual values available post-execution. We have already seen a QEP when describing the real-world problem pattern in Figure 2. The described problem pattern is much easier identified and fixed through the use of QEPs, since they provide a much more human-readable translation of the optimizer decisions made. The fix applied in Figure 2 was done by experts, and is precisely the process DistGALO aims to automate.

Let us now examine an automatic fix of a problematic QEP discovered and fixed by DistGALO. Figure 3a depicts the optimizer selected plan, and the DistGALO selected variant in Figure 3b. Let's first examine the problem with the former, and then describe the fix applied in the latter. The optimizer selected plan in Figure 3a suffers from a hash join (**HSJOIN (#3)**) with an expensive table scan (**TBSCAN (#4)**) as its outer input. The expensive table scan is considerably I/O intensive and accounts for 99% of the cost of the **HSJOIN (#3)**. The compiler knows it can utilize big block I/O for the table scan, which tends to be more efficient, and results in the cheapest overall access plan with a total cost of 160, 797 timerons.

Let us consider the DistGALO chosen QEP shown in Figure 3b. Note the reason for this plan not being chosen is solely due to the higher total cost of 177, 056 (**NLJOIN (#2)**). The cost of the nested-loop join (**NLJOIN (#3)**) makes up for the majority of the cost so perhaps the secret of the performance boost lies within. First, note that the inner and outer input streams of the join are both obtained from **FETCH-IXSCAN** operators **(4)** and **(6)**. This means that both inputs into the nested-loop join are ordered, and thus have better buffer pool exploitation. Secondly, the cost estimation tends to be quite pessimistic towards the I/O in nested-loop joins, further compounded by the partitioned indexes **DD_DI** and **CS_SHIPDATE**. These oversights result in a significant $3X$ speedup in the DistGALO selected plan.

QEPs are fundamental in DistGALO as they give the ability to translate queries into graphs, which can be decomposed into subgraphs, traversed, and altered with ease. The ability to do so hinges on a well defined and established graph framework, and as such, we opted for the Resource Description Framework.

RDF. The Resource Description Framework (RDF) is an XML based standard for describing data on the web, developed by the World Wide Web Consortium (W3C) [1]. RDF uses Uniform

[1] https://www.w3.org/RDF/

Resource Indicators (URIs) to link varying XML objects together without the need to embed them into one another as you would in XML. Due to the verbose nature of URIs, RDF is typically not intended to be read by programmers, but rather by computers. The RDF format is comprised of statement triples, namely the subject, predicate (property), and object (value). These three pieces of information are arguably enough to describe any single bit of knowledge or data. The subject is the who or what of the statement, the predicate is an existing fact about the subject, and the object is the final descriptive element regarding the subject. Objects do not only have to consist of primitive data types but can also be URIs pointing to other subjects. The QEP's LOLEPOPs are translated into subjects, with its metadata values defined by predicates and objects. The parent and children of each LOLEPOP are also defined similarly, except their values reference other LOLEPOPs. For example, each LOLEPOP consists of an inner and outer input stream, represented by the *<http://DistGALO/planDetails/property/hasInnerInputStream>* and *<http://DistGALO/planDetails/property/hasOuterInputStream>* predicates respectively. These are analogous to the child nodes in a binary tree. The parent node, or output stream of the LOLEPOP is defined by the *<http://DistGALO/planDetails/property/hasOutputStream>* predicate, followed by the reference value of the parent subject. Its flexibility allows us to easily define and store QEPs obtained from the optimizer. All queries and sub-queries used throughout the system are fed through the RDF Transformation Engine which parses the optimizer provided explain files containing the QEP, into an RDF. It allows for easier parsing and manipulation of the graph-based QEPs. This further gives the ability to breaking down queries into sub-queries and creating templates to be saved in the knowledge base.

To query the RDF defined QEP, we utilize the SPARQL Protocol and RDF Query Language (SPARQL). Also supported by WC3, it is widely used to retrieve, and alter data defined by RDF. The queries themselves are similar to SQL in that they contain the **SELECT** and **WHERE** keywords and allow users to declare variables within the query. The main difference, however, is in the **WHERE** clause, which is comprised of a series of triples, similar to RDF. These triples can, however, be used to declare variables, prefixed with the *?* or *$* character. For instance, *?x pref:diameter ?diam;* would query any triple with the *pref:diameter* predicate, store the resulting subject in *x*, and the object in *diam* which can be used in the projection of the data.

In this work, we utilize the SPARQL Jena Framework[2] to apply and make use of any required ACID transactions in Java. In addition, we utilize the Apache Jena Fuseki[3] web server as the base for the knowledge base responsible for storing and retrieving templates.

Optimization Guidelines. Guidelines are recommendation rules made to the optimizer during the compilation stage of a query. Three types are made available by the Db2 optimizer: *general*, *query rewrite*, and *plan* optimization guidelines. *General* optimization guidelines are used to set general-level optimization parameters *Query rewrite* optimization guidelines can apply alterations to the rules during the query rewrite stage. The third and only guidelines utilized in the DistGALO system are the *plan* optimization guidelines. These are applied during the third tier plan re-optimization step, and apply recommendations to the access

[2]https://jena.apache.org/index.html
[3]https://jena.apache.org/documentation/fuseki2/index.html

```
<OPTGUIDELINES>
  <NLJOIN>
    <NLJOIN>
      <IXSCAN TABID='Q1' INDEX='DD_DI'/>
      <IXSCAN TABID='Q3' INDEX='CS_SHIPDATE'/>
    </NLJOIN>
    <IXSCAN TABID='Q2' INDEX='SYS_SQL1'/>
  </NLJOIN>
</OPTGUIDELINES>
```

Figure 4: Plan optimization guidelines applied to access plan in Figure 3b.

methods or join types that will be chosen. Any invalid or unspecified sections of the query will be determined by the optimizer in the normal cost-based approach. Specifics regarding an access can be requested, such as a table scan, index scan, or list prefetch. Join requests can also be specified, including nested-loop, hash, and merge joins. Optimization guidelines are XML documents with *OPTGUIDELINES* as the root node, all requested operators as intermediate children and specified base tables/indexes as the leaf nodes. For example, the guideline in Figure 4 is used to apply the specified recommendations to map the optimizer selected plan in Figure 3a to the DistGALO selected plan in Figure 3b. The guideline specifies, read from leaf to root, that *Q1* and *Q2* are to be scanned using the **DD_DI** and **CS_SHIPDATE** indexes respectively. The results are used as inputs to a nested-loop join specified by the *<NLJOIN></NLJOIN>* tags, and show as **NLJOIN (#3)** in Figure 3b. The output is then also nest-loop joined with *Q2* indexed by **SYS_SQL1** and the result is finally obtained. This XML document can be appended to the SQL query statement, and applied by the optimizer without any user intervention. This was the method of choice when applying recommendations to user queries since it can be accomplished automatically, and provide a seamless process from a user perspective.

Apache Spark With the growing trend of increased data availability and storage, distributed data processing has also followed the trend. To mend the demand, several distributed frameworks have emerged, with (Apache) Spark[37] as one of the most popular[38]. Spark has many built-in solutions for common-use applications like streaming, machine learning, and SQL queries. It leverages the MapReduce[14] paradigm in combination with resilient distributed datasets (RDDs) that are used to retain read-only data across multiple nodes. Typically all data meant for distributed computation is stored in RDDs in key/value pairs. These RDDs then expose programmers to Spark's rich API to apply transformations on the data, like *reduceByKey()*, *mapValues(@f)*, and *sortByKey()* to name a few.

One of the bottlenecks in distributed computing is the communication cost between nodes. Great efforts are made to ensure that machines are located in close proximity with fast network speeds between each other, but even despite such efforts, it remains a big obstacle in distributed computing. If alike data can be grouped on a per-node basis, then the amount of data shuffling that will occur between RDD transformations can be reduced. Spark provides the control to specify which key/value pairs should appear together on a given node. Spark provides seamless fault tolerance since nodes chance to fail, and thus cannot guarantee users to specify a specific node for execution. Spark also provides built-in partitioning strategies that give the user some control over how RDDs should be distributed amongst the nodes. Two such partitioners exist, including *Hash Partitioners* and *Range Partitioners*.

The former using the hash of the key to determine a partition, and the latter allowing buckets with ranges to allocate data distribution. Additionally, Spark gives the ability to create *Custom Partitioners*, allowing programmers to utilize domain-specific information to partition the data. This is something we make heavy use of in this work since we want to ensure there is an overall balance between nodes, so as to not create skew across the cluster. More detail on the domain-specific application of partitioning can be found in Section4.3.

3 SYSTEM OVERVIEW

The previous GALO[11] system contained three major components: *Learning Engine*, *Matching Engine* and the *Transformation Engine*. DistGALO continues to embody similar architecture, but incorporates a revamped *Distributed Learning Engine* consisting of a *Distributed Learning Component* (DLC) and a *Random Sub-query Abridged Clustering Executor* (RSACE) module which includes several pruning strategies. A high level of the *Distributed Learning Engine* can be seen in Figure 5 with emphasis on the DLC. The Ranking Module, Template Creation, and process for populating the RDF Knowledge Base largely remain unchanged from GALO. The DLC can further be broken down into the following elements:

(1) *Distributed Sub-query Generator*;
(2) *Query Partitioner*; and
(3) *Distributed Sub-query Executor*;

Distributed Sub-query generator This component is responsible for processing daily query workloads and generating sub-portions of the original queries, known as sub-queries. After a query is processed, it is executed and the optimizer's chosen QEP is extracted. Using GALO's *Transformation Engine*, the QEP is mapped into its RDF graph representation, parsed, and subdivided into smaller sub-graphs. The size of the sub-graphs is based on the number of joins, set by the user. The sub-graph we term a sub-query projects the local and join predicates from the original query. The local predicates are used to generate new values by querying the database with dynamically sampled probing queries. These newly generated alternate predicate subqueries we henceforth termed, optimizer sub-queries, or Φ for conciseness. Using the Random Plan Generator (a DB2 built-in tool), a predetermined number of guidelines are generated and attached to each of the optimizer sub-queries to create a new set of sub-queries. These sub-queries are identical to their optimizer sub-query counterpart, with the exception of the attached guideline. The attached guideline provides an alternate execution plan that may otherwise not be considered during the cost-based plan evaluation. This new set of sub-queries are termed random sub-queries or Ψ, and its size is a factor of the number of random guidelines generated, but typically tends to be much larger than Ψ. DistGALO generates sub-queries in parallel using Spark, in contrast to the sequential generation of GALO. The workload queries are partitioned based on several different metrics. At this stage, both sets Φ and Ψ can be combined and distributed throughout the cluster for processing. This, however, turns out to be a less effective approach due to the relationship between the sets (more detail in Section 4). Thus, after the sub-query generation step, both the Φ and Ψ sub-query sets are passed along the pipeline to their respective executors.

Query Partitioner The query partitioner is a swiss army knife for partitioning three different kinds of queries. The first is the initial query workload that is split into the respective

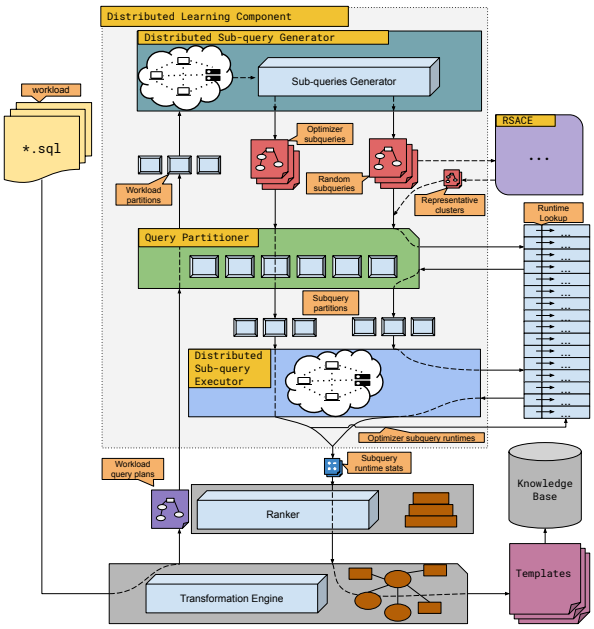


Figure 5: System architecture of DistGALO's Distributed Learning Engine.

sub-queries. Four different partitioning strategies are available to choose from: hash, cost, join, and weighted alternatives. Φ is another set of queries, that require their own partitioning strategies. These only include the hash and cost partitioners. The last type of queries the Query Partitioner must handle, arguably the most important, since it's proven to be a bottleneck, are the Ψ, or random sub-queries. These employ four strategies as well: hash, cost hybrid variant, exchange, and runtime partitioners. Each of these will be described in further detail in the following Section 4. It's important to note that these are options available to the user, which can be set from the main configuration.

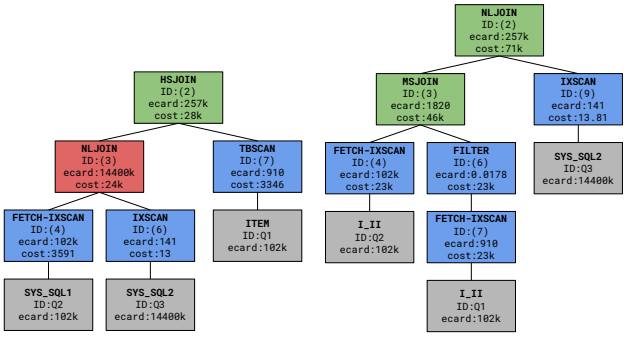
Distributed Sub-query Executor This component is similarly hard to generalize for every time of query and will, therefore, be segmented in two. The first is the execution of the optimizer sub-queries, and the second is responsible for execution of the random sub-queries. The execution order of the two is also important and is done so in the order presented.

First, the Φ set of sub-queries is processed. Initially, the Query Partitioner distributes this set of queries to all nodes whereby each is processed. The processing of each ϕ sub-query involves generating their QEPs, and executing each in order to obtain the runtime and runtime statistics. Based on the metrics obtained, all runs are ranked by the *Ranking Module*, with the winner being saved in the HDFS for future reference. A time lookup table with all the best-ranked execution of each ϕ is generated to provide upper-bound thresholds for the processing of Ψ.

Following is the processing of the Ψ set of sub-queries. This is a critical step, previously proving to be a bottleneck in GALO, due to the large number of sub-queries generated. DistGALO now tackles this stage very strategically, from the partitioning strategies to pruning approaches to remedy the massive search space. The time lookup table from the previous step is also heavily utilized to ensure that ψ sub-queries are not run for longer than they need to be. Several partitioning strategies are also made

available, each with its strengths and trade-offs. Similarly, sub-queries are processed in a similar fashion to Φ, and so all top-ranking sub-queries are ultimately saved in the HDFS.

Random sub-query abridged clustering executor (RSACE) RSACE was devised as a pruning strategy to provide further speedups with minimal impairment to the objective of the Learning Engine. The module is a feature optional to the user, that can be toggled on or off on demand. This strategy is only applicable to the execution of Ψ since it has the prerequisite of a time lookup table from the execution of Φ. The process involves vectorizing the predicates of each ψ, grouping by their respective QEP structures, applying clustering techniques within each group, and executing representatives from each cluster. The module can toggle to account of any false positive representatives that may have arisen. Potential false positive representatives' clusters are processed using the Distributed Sub-query Executor to ensure the DLC is still effective in the templates it saves in the Knowledge Base.



(a) Plan selected by the optimizer. **(b) Plan chosen by the DistGALO system.**

Figure 6: Problem pattern with expensive nested-loop join and DistGALO discovered fix.

4 DISTRIBUTED LEARNING

The Distributed Learning Engine is the more complex component of the DistGALO architecture as a whole. It is further subdivided into several stages: *sub-query generation*, *optimizer sub-query execution*, *random sub-query execution*, and *partitioning*. There are intermediate steps involved that are mentioned and described in limited detail since they are not the main emphasis of the distributed learning process. These include generating QEPs with actual cardinalities, ranking, preparing queries, problem pattern creation, and managing input/output files from/to the HDFS. The latter is used as a common communication source for all nodes throughout the stages of execution. Nodes should have the most up-to-date state of the system, thus the HDFS is periodically updated in-between module executions, but only with essential elements to avoid redundancy and reduce size. These steps are less computationally intensive and are thus omitted from diagrams and most discussions. The main focus will be limited to the main processes mentioned above as they are the most computationally and resource-heavy.

The output of the Distributed Learning Engine in DistGALO remains the same as that of GALO. It aims to discover the more optimal, less resource consuming, faster executing, random sub-queries, in comparison to their optimizer sub-query counterpart. Let us consider one such case depicted in Figure 6 where we observe a problematic QEP chosen by the optimizer, and the appropriate fix selected by DistGALO; let's examine each independently. The optimizer selected plan in Figure 6a first joins *Q2* with *Q3* using nested-loop join **NLJOIN (#3)**. It then joins the results with *Q1* using the hash join **HSJOIN (#2)**, for a total cost of 28, 860 timerons. Note that by joining *Q2* with *Q3* first, it leads to an expanding join, with cardinality of roughly 14million. This expanding join results in the majority of the cost, with 24, 902 timerons.

Consider the DistGALO selected plan in Figure 6b. Its total estimated cost of 63, 682 is more than double than that of the optimizer, yet exhibits a 4*X* performance speedup. Let us proceed by analyzing the QEP inner-workings and determine the optimizer oversight. First note the join order difference, whereby *Q2* is first joined with *Q1*, then finally with *Q3*. The former join is performed by what seems an expensive **MSJOIN (#3)** with a cost of 46, 019 timerons. Note, however, that this join, despite being the most costly, reduces its inputs to an estimated cardinality of 1, 020 in contrast to the expanding **NLJOIN** with 14 million in the optimizer selected plan. The inner and outer inputs utilize **FETCH (#4,#7)** operators, each with a cost of 23, 000 timerons. A crucial observation is that the index, **I_II** used by both to fetch row ids, is the exact same. This means that during the merge join operation, whilst the outer is being read, the buffer pool is heavily utilized for the inner with minimal I/O. Effectively the cost of the **MSJOIN (#3)** is halved, propagating the speedup to the following **NLJOIN (#2)** above. This oversight is largely attributed to the cost model since is assumes I/O must be performed. These types of modification provide insight into the ever demanding need for domain experts, and more-so automated systems like DistGALO.

4.1 Cluster and Spark Configuration

Cluster To remedy the lengthy runtime of the GALO Learning Engine, a distributed cluster was used to improve scalability, runtime, and to fully utilize the hardware available. Computations were performed on the SOSCIP Consortium's Cloud Data Analytics computing platform(s). SOSCIP is funded by the Federal Economic Development Agency of Southern Ontario, the Province of Ontario, IBM Canada Ltd., Ontario Centres of Excellence, Mitacs and 15 Ontario academic member institutions [3]. The SOSCIP platform ensures that nodes are free of user traffic, thus minimizing any noise that would otherwise be present in multi-user nodes. All nodes are located on the same internal network, so communication overhead between them is minimized and limited to the partitioning strategy employed by the system. The cluster is comprised of 9 nodes, including a master, each with 4 virtual CPUs, 16GB of RAM, and 120GB of disk space. To effectively use the cluster, we used the general-purpose cluster computing platform, Apache Spark. The open-source platform extends the MapReduce model to support a large variety of computations including SQL queries, which fit the DistGALO system requirements closely. It has also proved to be quite effective in the community and has withstood the test of time, and was therefore selected as the tool of choice. We also utilized the Hadoop Distributed File System (HDFS) to seamlessly share data between all the nodes. Great effort was made to minimize the number of reads/writes to the HDFS since it could pose as a potential bottleneck due to the large amount of files and data involved. At the time of writing, and in the latest iteration of DistGALO, Apache Spark 2.4.0 and Apache Hadoop 2.7.7 were used.

Table 1: Apache Spark Cluster Configuration

Spark Property	Value
spark.cores.max	36
spark.executor.memory	4G
spark.executor.instances	36
spark.executor.cores	1

Spark Configuration Apache Spark's configuration is quite extensive and allows for very flexible and versatile tuning of the cluster. Some of the available configurations available include setting: number of cores per executor, memory per executor, total cores available to the cluster and so on, see[4] for a comprehensive list of Spark's configuration options. The cluster used includes 9 nodes, one acting as the master and worker, and the remaining 8 just as workers. The cluster has 36 virtual CPU cores, and 144GB of RAM available. Since the system is very precise in how it must be executed, no automatic methods exist for finding and setting the most optimal configuration. There are also various variables that such a system would not be able to tune; including the database size, database configuration, and query workload, just to name a few. We have thus opted for the arduous task of experimentally finding the most optimal configuration. Table 1 displays the Spark configuration settings experimentally found to be best. The philosophy behind each is as follows. We limit the number of cores utilized per executor (*spark.executor.cores*) to 1 to maximize concurrency on each node. Since each node has 4 cores available we expect 4 executors to be running concurrently, thus limiting the RAM to 4*GB* per core (*spark.executor.memory*). Finally, since we have 9 available machines with 4 cores each, we limit both the number of executors (*spark.executor.instances*), and the total number of cores (*spark.cores.max*), to 36, since those are the total available VCPUs available in the cluster. These configuration settings are based on several properties that were measured in-house, primarily runtime and resource utilization. A common symptom and bottleneck of distributed systems is skew. This typically occurs when a small subset of the nodes available takes longer to complete the task, thus halting the whole workflow including the nodes that have already completed. This is something that Spark tries to address under the hood, but when skew continues to be evident with the default settings, more customized settings are required. Due to the non-deterministic nature of executing randomly generated execution plans, skew is quite prominent and must be addressed directly. We thus make use of several characteristics and heuristics from the generated sub-queries in an effort to distribute the workload evenly throughout the cluster.

4.2 Sub-query Generation

The sub-query generation process takes an SQL query as input and creates various sub-queries, including ones with optimizer chosen execution plans, and others with a randomly generated execution plans. Once the two batches of optimizer and random sub-queries has been generated, they are passed on to the execution step of the pipeline. Let us delve further into the sub-query generation process. The process begins with a set of queries(workload) to be learned from, $Q = \{ q_1, q_2, ..., q_n \}$ for n input queries. Each query q_i is further broken down into a set of sub-queries $S_i = \{ s_{i1}, s_{i2}, ..., s_{im} \}$ where m sub-queries are generated for a given query q_i. The generation of sub-queries for a given query is based on the execution plan generated by the Db2 optimizer. Given a sub-query of size J joins, every combination of J join sub-graphs are generated by $subgen(q_i, J)$ [11]. The sub-query s_{ij} thus represents the j^{th} query generated from the original q_i^{th} query. These set of sub-queries can be more compactly be represented by $\mathbb{S} = \{ subgen(q, J) \mid q \epsilon Q \}$. For each sub-query s_{ij} a set of k alternate predicate sub-queries $\{a_{ij1}, a_{ij2}, ..., a_{ijk}\}$ are generated by the algorithm $altgen(s)$ [11]. These are all modifications of the original sub-query s_{ij} but where, through sampling of the attributes, have different local predicates. For instance, a_{ij1} and a_{ij2} would be the exact same query but would have different predicate values. These sets of alternate predicates can too more succinctly be represented by $A = \{ altgen(s) \mid s \epsilon \mathbb{S} \}$. To summarize, the alternate predicate sub-query a_{ijk} thus represents a sub-query formed from query q_i, with the j^{th} sub-query, and the k^{th} alternate predicate sub-query. Each alternate predicate sub-query then forms a group $g_{ijk} = \{ \phi, \psi_1, \psi_2, ..., \psi_q \}$ where ϕ represents the optimizer chosen query execution plan, and ψ_q is the q^{th} randomly generated query execution plan for any given alternate predicate sub-query a_{ijk}. To simplify the nomenclature, if G were the set of all groups g_{ijk} and a flat-map like function were applied, then grouping all optimizer chosen plan sub-queries into a set Φ and grouping all the random plan sub-queries into the set Ψ. If the sub-query generation process were observed to be a black box, the input would be the set of queries Q, and the output would consist of the sets Φ and Ψ.

The sub-query process is fairly computationally intensive since tables must be sampled, and a fair number of files are created as a byproduct. The process was therefore distributed among the nodes in the cluster to fully utilize all available resources. Each query in Q is first executed to obtain the actual runtime and cardinalities as opposed to the just the estimates. Following this, various attributes of the queries were used to effectively split them into smaller workloads to be sent to individual nodes. The heuristics aimed to capture the characteristic where more complex queries typically result in a longer runtimes of decomposing them into sub-queries. Larger queries (more tables and joins) correlate to large possible combinations between joins, and thus a larger number of sub-queries. This alone provided a gateway of grouping equally balanced workloads, which maps to the mechanics of the Join Partitioner. To further refine this methodology, the estimated cost, or timerons, were taken into account in conjunction with the Join Partitioner. Each metric was given its own weight, and ultimately provided the most informed decision a partitioner could make, resulting in the Weighted Partitioner. Once the work is distributed along to the nodes via the partitioners, the HDFS is populated with both the optimizer selected QEP sub-queries Φ and the randomly generated plan sub-queries Ψ. These are then passed down into the execution stage of the pipeline.

During the sub-query generation process, it is possible to derive combinations of joins between tables that were not present in the original table. For this reason, there is a lack of information pertaining to any estimates that would otherwise be available via the query compiler. In this event, a probe is made into the data to get an estimate of the missing pieces of information, like actual cardinalities. We only consider it an estimate since doing a full query of the join predicate might be too costly of a task and so only a sample of the data is considered. Rather than read a

[4]https://spark.apache.org/docs/latest/configuration.html

sample % of the pages, we take a more straight forward approach that scales with the size of the tables. That is, we only consider an upper limit, or cap, to the number of pages that will be read, regardless of the size of the table. This means that there is a greater inaccuracy with an increase in the difference between the actual number of pages and the cap. The upside is that this method is very efficient and does not bottleneck sub-queries that involve large tables. This potentially growing margin for error is also acceptable in the context of sub-query generation since the resulting cardinality is not mission-critical and allows for such a margin. It may ultimately lead to some template ranges that do not accurately describe their cardinality ranges, but this has experimentally shown to be a worthy trade-off.

The sub-query generation step is the second-longest executing in the learning process, and also in GALO (Figure 1). This provided sufficient motivation to also distribute this process but required some analysis of several nuances. First, the amount of workload items that can be distributed is significantly less in comparison to the number of random sub-queries. This leaves less room for shuffling queries around, and thus don't expect as great a benefit from the distribution process. Secondly, not all partitions used in the distribution of random sub-queries are applicable since less information is available being an earlier stage of execution. The runtimes, costs, and relationships from the groups of sub-queries allowed certain information, like the bounding thresholds from optimizer sub-queries, to be used as metrics in the partitioning process. These kinds of relationships and information are not existent during the sub-query generation stage, so different techniques must be devised to intelligently partition the queries.

The Cost (hybrid), Exchange and Runtime Partitioners utilize the previously mentioned information and relationships, so are deemed ineffective in the context of sub-query generation. In this context, we take each indivisible work item, the input query, and generates all possible combinations from the number of joins and tables involved in join predicates. The relationship observed is that larger queries, ones that have a larger number of joins and tables, exhibit a longer sub-query generation process. The Join Partitioner utilizes this information precisely, by balancing the workload only based on the number of sub-queries expected to be generated by the input query. This one shortcoming of this approach is that all heuristics about the complexity of the query are ignored. Table cardinalities, expanding joins, and any other statistical information are disregarded. To mend this, the Weighted Partitioner uses both the information of the number of sub-queries, but also all heuristic information that's packaged into the cost, into one partitioning metric. This hybrid approach seems to be the most informed and thus the best performing partitioner, as will be shown in the evaluation sections of this work.

4.3 Partitioning

Partitioning in distributed systems is a crucial element in ensuring the cluster is working at its optimum. In most cases of distributed computing, the main bottleneck tends to be the communication cost between nodes, or network traffic. When performing transformations on data, shuffles of data are triggered across the cluster, incurring communication cost and slowing down overall runtime. If alike data is however grouped, transformations would cause less shuffling across the network, thus reducing communication cost. Partitioning provides a grouping of keys based on some function, either built into or defined by the user (in the context of Apache Spark). In other words, partitioning gives the ability to preemptively designate how keys are to be grouped when allocated to nodes. So a user has the control to determine if two keys should end up on the same node, though cannot guarantee that they will not. Typically the effectiveness of partitioning comes from reducing data shuffle during the transformations applied, but DistGALO only does a single mapping followed by a single reduction. How then do we benefit from partitioning, and is it significant enough to warrant a discussion? DistGALO's main goal is to execute a large number of sub-queries with varying complexities. More complex queries usually take a longer time to execute, and so grouping many complex queries on a single node would case some skew. So the benefit of partitioning comes from the initial sub-query allocation, as opposed to the reduction of data shuffling. The initial distribution dictates which nodes will be overloaded, and which will sit idle waiting for the rest to finish. This can be achieved through partitioning and careful analysis on how to partition sub-queries together. Let us explore the varying partitioning strategies made available through Apache Spark, and others that are proposed in this work.

Apache Spark comes equipped with three options when selecting a partitioner. The first is a Hash Partitioner, which groups data based on the hashed key values. This is a viable option in some scenarios so it is worth further discussion, which can be found in the section to follow. The second partitioning option is the Range Partitioner, which partitions based on the natural ordering of keys, and places keys in predetermined ranges. This could allow us to possibly group similarly complex queries together, which provides the opposite of the desired effect, which is to balance the overall complexity among the partitions. This we deemed not a viable option and is further excluded from the discussion. The last option that Spark provides, is the ability to write a Custom Partitioner. This gives the user the ability to customize precisely how keys are grouped within the context of their application. In this work, we have devised several different partitioning methods, a Cost Estimate Partitioner with a hybrid alternative, a Runtime Partitioner, an Exchange Partitioner, a Join Partitioner, and a Weighted Partitioner. Not all partitioners are applicable to every distributed part of the system. Some metrics used (joins, weights, runtime) are unique to the component they are utilized in, since the information required may only be available during that stage of execution. Our evaluation has identified the optimal choice of partitioner for both the sub-query generation and sub-query execution stages.

Hash Partitioner The Hash Partitioner groups keys based on the hash value of each key, so keys with the same hash values end up in the same partition. More formally we can write this as: $partition = getHash(key) \,\%\, numPartitions$, where $getHash$ is a built-in function which generates a hash value, and where $numPartitions$ are the number of partitions set by the user or automatically by Spark. Given a hash function adhering to uniformity, we expect that the resulting partitions would have an equal number of key/value pairs assigned to each. This is a desired quality in some parts of DistGALO, like in the optimizer query execution phase of the system. This approach is advantageous in that it requires no prior knowledge about the key/value pairs and therefore has little overhead. There are scenarios in which an even distribution is not as desirable, and may hinder the runtime. In the context of SQL queries, the hash partitioner doesn't have any information regarding the complexity of the queries. This oversight may result in complex queries being grouped on the

same node, leaving the less complex and completed nodes to sit idle. We thus want a more quantified way of assessing the complexity of a query to create an even distribution of workload complexity amongst all nodes.

Cost Estimate Partitioner We turn to a heuristic-based approach to partition the sub-queries based on some estimated metric of complexity. Db2 offers such a heuristic in the form of an estimated cost in timeron units. A timeron is an estimate of the total I/O and CPU cost the Db2 manager might incur during execution. The estimate is derived from table statistics, indexes, predicates involved, cardinalities and other variables. A great advantage this approach offers is that the cost estimate values are available at compile time. That is they do not require execution of the sub-query, which is a timely and costly procedure. Timeron values are obtained from the QEP that the Db2 optimizer generates through the explain facility.

input : The set of sub-queries to be distributed $\mathbb{S}$
output: The partitions with sub-queries assigned **filePartitions**

1 Let $filePartitions = Map(query, partition)$;
2 Let $partitionBuckets = [0] * numPartitions$;
3 **for** $s \in \mathbb{S}$ **do**
4 $\quad minIndex$ = getMinBucketIndex($partitionBuckets$) ;
5 $\quad partitionBuckets[minIndex]$ += $s.metric$;
6 $\quad filePartitions$.put(s, $minIndex$) ;
7 **end**
8 return $filePartitions$;

Algorithm 1: Algorithm for partitioning of sub-queries

The Cost Estimate Partitioner is a Custom Partitioner that groups queries based on their total estimated cost timerons. The prerequisite is that all cost estimate values must be available for each sub-query to be partitioned. This means that every sub-query in question must have their QEP generated by the DB2 explain facility. The initial partitioning process works by creating a bucket for each partition, an attribute set by the user. The queries are then sequentially evaluated, and their cost estimated values are used to populate the buckets. The assignation is as simple as assigning the current cost value to the lowest bucket. Algorithm 1 shows the trivial, yet effective assignation of sub-queries to partitions. First the *filePartitions* are initialized (line 1) with an empty map, whereby the keys are the sub-queries, and the values, the corresponding partition the query is to be put in. The empty set of buckets is then initialized (line 2) and will keep track of the total cost for each partition assigned thus far. For each sub-query, the current smallest bucket is obtained (line 4), updated with the query metric (line 5), and set to the corresponding partition (line 6). The query metric can be any characteristic of a given sub-query. In the case of the Hash Partitioner, the metric is the estimated total cost obtained from the optimizer.

Though there is quite a large variety of cost values, when dealing with a large number of sub-queries, the partitions are eventually able to flatten out and more or less have the same values. Optimally, each partition would have the same cost values, but in practice is not the case, though relatively close. Since the estimated costs relate to the complexity of the queries, having partitions with an overall equal level of complexity, translates to partitions that must finish execution at roughly the same time. This too in practice is not always the case since we are relying on estimates, but in general, it is quite effective in its predictions. The accuracy of the Cost Estimate Partitioner is largely dependent on the accuracy of the estimated cost values. The estimates are constantly being improved by the Db2 optimizer team, and with more and more accurate cost values, the Cost Estimate Partitioner benefits just as equally.

When partitioning random sub-queries we can leverage some prior knowledge to devise a hybrid cost approach. Each random sub-query has a corresponding optimizer sub-query which also has an associated cost. The latter acts as an upper bound to the former and so the cost can analogously be used to do the same. Either the threshold τ will halt execution, in which case we favour the cost of ϕ, or random sub-query will successfully execute, thus favoring the cost of ψ. We can summarize this formally with $min(\phi_c * \tau, \psi_c)$ where ϕ_c is the cost of the optimizer sub-query, and ψ_c is the cost of the random sub-query. The multiplication of τ is the execution threshold ratio, used to account for the threshold % used when creating the upper-bound thresholds. The calculated metric is ultimately what is used as the query $s.metric$ in Algorithm

Runtime Partitioner Instead of using estimated timeron values, a more reliable and accurate approach is to use the runtime values. Runtime values refer to the exact CPU time taken to execute a given query. We must emphasize that timerons are not the same as runtimes, since the former is an estimate, and the latter an exact value. The algorithm used is identical to the Cost Estimate Partitioner (Algorithm 1), but with a different query metric. Instead of using the timeron values, we simply swap them out for the actual runtime values. The process of filling the buckets works identically, by sequentially iterating through the queries' runtime values, and assigning each to the minimum valued bucket at the time of execution. The result is a set of partitions each assigned their own set of queries, and a partition value depicting the exact runtime it will take for it to complete execution of its assigned queries. The value of all buckets should be as close as possible to ensure partitions are completed in as similar times as possible. Partitions are later subdivided into multiple executors running concurrently on each node. It should be noted that the partitioning stage precedes the query execution stage, so using runtimes to partition seems paradoxical. This is indeed true if we are partitioning the same queries being executed, but in general, is not the approach taken when using the Runtime Partitioner. This requires a more in-depth analysis into the queries being executed, this is further elaborated upon in Section 4.5. To show the effectiveness of the Runtime Partitioner, an experiment comparing all partitioners was conducted and further described in Section 5.2.

Exchange Partitioner The exchange partitioner attempts to utilize all available data to create a prediction of what the random sub-query execution runtime might be. First let's summarize the available information at this stage in execution. The optimizer sub-query has at this stage been processed and provides its cost in timerons and runtime in seconds as information. The sub-query to be executed only contains the cost in timerons, but without a runtime. For simplicity we will discuss this process in two separate steps that will later be combined.

The first is calculating an exchange rate from the optimizer sub-query that can be used to calculate an estimated runtime using the random sub-query's cost estimate. This we can formally write as $\psi_r = \psi_r(\frac{\phi_r}{\phi_c})$ where ϕ_r, ϕ_c are the optimizer sub-query's

runtime and cost respectively. Similarly, ψ_r, ψ_c are the random sub-query's runtime and cost respectively. More precisely ψ_r is the now estimated runtime of the random sub-query to be executed.

The second part involves calculating the upper bound limit that will be set on the random sub-query's execution. If this limit is reached, the execution is terminated, and the random sub-query is no longer considered as a candidate.This can formally be written as $C\lceil \frac{\phi_r * \tau}{C} \rceil$ where τ is the execution threshold ratio, and C is a constant threshold step minimum enforced by Db2. At the time of writing this paper, this is set to 10 but is subject to change with the ongoing development of Db2.

Putting it all together we have the predicted runtime of the random sub-query, and the upper bound limit and thus one of two things will happen. The query is under the threshold limit and completes successfully, which would ideally be close to the predicted runtime ψ_r. Alternatively the runtime will reach the threshold limit whereby the execution will stop. We can finally combine them into the following heuristic: $min(\psi_r(\frac{\phi_r}{\phi_c}), C\lceil \frac{\phi_r * \tau}{C} \rceil)$. The minimum of both will dictate which of the events is likely to happen, and can therefore be used as the query $s.metric$ (line 5) of Algorithm 1 for the Hybrid Cost Estimate Partitioner.

Join Partitioner In some scenarios, like the sub-query generation stage, we take a unique approach to partitioning the queries. Note that at this stage we are not yet partitioning sub-queries, since those have not been generated yet, but are instead partitioning the initial workload queries. The goal remains the same as with the previous partitioners, which is to reduce skew by balancing the workload among the nodes as evenly as possible. To do that, we must consider which attributes the queries drive complexity and thus execution time. One such attribute is the number of sub-queries we expect to be generated by a given query. This can easily be calculated beforehand since the sub-queries are comprised of generating k-sized sub-queries from a set of N tables queried by the original query being decomposed. This is equivalent to finding the possible k-combinations from a set of N elements, or more formally $\binom{N}{k}$, where N is the number of tables, and k is the size of the sub-query.

The calculated value represents the expected number of sub-queries that will be generated for a given query. The sub-query generation is done using RDF, so the complexity of creating and parsing the RDF graph is directly correlated to the number of joins or the number of expected sub-queries. Each query will thus have this projected number of sub-queries value associated with it, which is precisely the metric used in line updated with the query $s.metric$ (line 5) of Algorithm 1.

Weighted Partitioner In a continued effort to try and find a heuristic for complexity and runtime regarding sub-query generation, a hybrid approach seemed appropriate. Not taking the timerons into consideration is neglecting years of knowledge and effort of Db2 development aimed at predicting the cost of a query. This heuristic, paired with the Join Partitioner combines both pieces of crucial information to most accurately predict overall runtime of generating all possible sub-queries from a given query. This we achieve by attaching a weight to both, the number of expected sub-queries, and the cost of the query being decomposed. More formally this can be written as $(\binom{N}{k} * w_j) + (q_c * w_c)$ where w_j and w_c are the join and cost weights respectively and q_c is the cost of the query. These weights have no set value, other than what has proven to be the most effective experimentally. The calculated value is similarly used as the query metric (line 5) of algorithm 1.

4.4 Optimizer Sub-query Execution

The next stage involves running the optimizer chosen query execution plan sub-queries described by Φ in Section 4.2. This means that each sub-query must be executed by Db2 to obtain the actual runtimes. As this is a lengthy and computationally heavy step, the choice of distributing the work among the cluster was made.

The next and arguably most important step is the choice of partitioning strategies described in Section 4.3. The options, however, are limited to only two, as runtime partitioning is not available pre-execution of sub-queries. The objective is to obtain the runtimes from each sub-query, so the partitioning strategies are narrowed down to either cost estimate, or hash partitioning. As previously observed, the hash partitioning strategy does not offer much intelligence in the distribution of sub-queries between partitions. It does not take into account any heuristics and instead simply distributes the sub-queries as evenly as possible throughout the partitions. Conversely, the cost-based partitioning strategy offers insight into the complexity of the query and is a decent heuristic to group low cost, fast-running queries together, while allowing the larger ones to run independently, and ultimately avoid skew. This makes a strong case for the latter, however as previously mentioned, there is a prerequisite to running using a Cost Estimate Partitioner. The estimates must be available, which requires the execution of program *db2exfmt* on each query. This ensures that the QEP for the query is generated, and though it is not as costly as actually running the query, it does add some overhead. The overhead would need to be evaluated in order to determine if the added time is less than the difference between the hash and cost partitioner execution time. The choice was made easy however since most of the random sub-query partitioning strategies actually require the costs of their optimizer sub-query counterparts. This means that the overhead could, in general, could not be avoided and was a cost that would be incurred anyway. We, therefore, opted for the cost partitioning strategy as our default for the execution of Φ.

There are scenarios in which the sub-queries do no run in a respectable amount of time, and so a global threshold was placed to ensure that if met, the query execution was cut off. This avoided some edge cases where a sub-query would run indefinitely thus halting the system since all other nodes wait until completion. Even though the environment used was not shared amongst users, the system is still susceptible to small amounts of noise which propagate and cause variations on query times. In an effort to reduce this noise, we have opted to run each query three times and obtain the best execution. We term the best run here as the one that ranks the highest amongst all other runs. This includes an additional ranking process, similar to that of GALO, that must be run after all three runs of each sub-query. The best performing run is kept and saved in the HDFS, while the remaining two are discarded. The ranking takes into account several statistics extracted from the optimizer, all of which are listed in Table 2. The ranking is based on weights and each query run is given a score. ELAP_T is given a weight of 50% and all the remaining properties are given the remaining combined 50%. The weight favours the elapsed time since that is the most critical element, and most costly in obtaining, therefore has the most

Table 2: Sub-query post execution ranking properties considered.

Abbreviation	Description
ELAP_T	Elapsed Time
BP_D_LR	Buffer pool data logical reads
BP_D_PR	Buffer pool data physical reads
BP_TD_LR	Buffer pool temporary data logical reads
BP_TD_PR	Buffer pool temporary data physical reads
BP_D_W	Buffer pool data writes
BP_I_LR	Buffer pool index logical reads
BP_I_PR	Buffer pool index physical reads
BP_TI_LR	Buffer pool temporary index logical reads
BP_TI_PR	Buffer pool temporary index physical reads
BP_I_W	Buffer pool index writes
U_CPU_T	Total User CPU Time used by agent (s)
SSH_HWM	Shared Sort heap high water mark
ROWS_R	Rows read

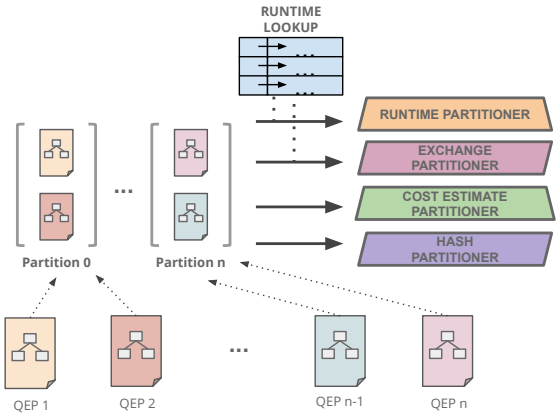


Figure 7: Partitioning process of random sub-queries using one of the available partitioners.

significance. The ranking process is described in greater detail in GALO[11].

4.5 Random Sub-query Execution

After the optimizer selected QEP sub-queries have completed, the random QEP queries are prepared for execution. This is similar to the process of executing the optimizer variant described in Section 4.4. Similarities include the need for execution to obtain runtime statistics and elapsed time, thus prompting the need for a distributed solution. There are some subtle differences, however. The first most significant observation is the sheer amount of random sub-queries in comparison to the optimizer sub-queries generated. The former exceeded the latter by about 7 times, on average. This figure does vary slightly since the queries generated are random and as such vary in number, but also because the queries in the workload also vary. The second observation that can be made is that some valuable information is now accessible from the execution of the optimizer queries, mainly the runtimes. This may not seem to be much of use, but we must first examine the relationship between Φ and Ψ described in Section4.2.

Recall the groups $g_{ijk} = \{ \phi, \psi_1, \psi_2, ..., \psi_q \}$ created during the sub-query generation phase, where i represents the original query number, j the sub-query number, and k the alternate predicate number. Within each group g there exists an optimizer chosen QEP sub-query or ϕ. In addition, there exists q number of ψ randomly generated QEP sub-queries. As a recap, the ultimate objective in creating a template is to discover at least two sub-queries with randomly generated QEPs, which are part of the same sub-query, but with varying alternate predicates. In other words, for two groups belonging to the sub-query, there must exist at least two ψ sub-queries that are better than their ϕ sub-query counterpart, in order for a template to form. At the group level, the only concern is for any given group g, to find a ψ that outperforms the ϕ. Any ψ that does not outperform the groups ϕ is discarded as it cannot possibly contribute to the creation of a template. The optimizer chosen QEP ϕ therefore governs the runtime for each ψ in the group thus describing the relationship between Φ and Ψ.

This relationship though not immediately evident provides three possible ways to provide significant speedups in execution of Ψ, without any sacrifice to the number of templates created. The first speedup involves utilizing the cluster to distribute all of the sub-queries in Ψ with a better, more informed, partitioning strategy. The second involves using thresholds to limit the execution time of the sub-queries. The last speedup involves a pruning method that removes unnecessary runs.

Partitioning

A summary of the available partitioning strategies for the random sub-query execution stage can be summarized in Figure 7. The figure only depicts two partitions for simplicity, and the four available partitioners described in Section 4.3. Let us review each individually:

A naive solution would be to divide the work up evenly on each node, which can be achieved with ease using the Hash Partitioner. The resulting state of the cluster would be an even distribution of queries amongst all nodes. The limitation to this, however, is that the complexity of the queries is not taken into account. Take for example a scenario where two nodes must each execute 3 queries, but by chance, the first node got assigned fairly simple queries that run fast and finish quickly. The second node, again by chance, as is often the case with distributed systems, is assigned 3 very complex queries, which take a significantly longer amount of time to complete, and have now introduced skew into our system. detail in Section**??**.

The second approach in partitioning Φ is the Cost Estimate Partitioner. The estimates are obtained through the QEP files, which are generated through the explain table format command, *db2exfmt*. These values tend to be higher with more complex queries, and can, therefore, be used to balance sub-queries throughout the cluster. The caveat is that there is some overhead associated with creating the QEP files. The files do not happen to be available at this stage of the execution process, so they must, therefore, be generated. This overhead though not very significant on a per-query basis when considering thousands of sub-queries, becomes quite significant. The overhead negates some of the speedup advantages the cost-based partitioning strategy provides, making it less effective. To compound this further, the estimates are not always reliable and can sum up to a significant amount of error. Both of these factors surmise to a considerable amount of skew re-introduced into the cluster, nullifying the purpose of a partitioning strategy. We must, therefore, seek an alternate partitioning strategy void of such shortcomings.

The third option for partitioning is using the actual runtimes, in contrast to the estimates. Ideally, the runtimes of the ψ queries would be used, but this is only available post-execution, which is the very task we sought to achieve. Consider however the actual runtime values of Φ that were executed in the previous step. These runtimes, in essence, dictate the upper-bound thresholds of their ψ counterparts. Let us recap, the original goal of partitioning is to ensure that all nodes complete execution as close as possible to one another. We have now established that we have the upper-bound execution time of each ψ query, and thus if we partition based on the upper-bounds, then we are achieved what we originally set out to do. This strategy doesn't rely on any estimates, so the expectation is that each node will run the exact time it was allotted. This generally holds true, with the exception where ψ sub-queries significantly undercut their upper-bound. This in practice is exhibited on each node and tends to balance out overall. The runtime partitioning strategy introduces no overhead since it only utilizes information already available, and is, therefore, the better performing of the three in this stage of the learning process.

The fourth alternative to partition the random sub-queries is the Exchange Partitioner. It utilizes prior knowledge of ϕ runtime information, ϕ costing, and the current ψ costing. An exchange rate of $\frac{s}{timeron}$ is calculated from the ϕ counterpart and then applied to the costing of ψ to derive an estimated runtime. The resulting time could, however, exceed the threshold that is set as an upper-bound to terminate long-running sub-queries. This upper bound is based on the ϕ sub-query counterpart runtime, multiplied by the threshold ratio, and rounded up to the next Db2 step threshold limit. This upper bound always dictates the runtime of the ψ sub-query unless it executes in less time. For this reason, the minimum of the two is taken and used as the heuristic when populating the buckets in the partitioner. The summary and more formal definition of this calculation is shown in Section 4.3. This approach still heavily relies on the cost estimates obtained from the query optimizer but is able to disregard cases, where there are inaccuracies in the over-estimates of the cost. Incorrect under-estimates still propagate via the exchange rate and are the likely culprit to any observed skew.

Thresholds Once the nodes have been assigned the group of sub-queries that must be executed, a brute force method of simply running the queries does not suffice. There are scenarios in which some of the randomly generated QEPs will not be valid, and will never truly terminate on their own. To mend this, a global threshold can be set, similar to the one outlined in Φ execution in Section4.4. This though somewhat effective does not perform too well due to the substantial number of ψ sub-queries. The above-mentioned relationship can, however, aid in making a threshold that caters to each group. A threshold can be created for each group $g_{ijk} = \{ \phi, \psi_1, \psi_2, ..., \psi_q \}$, such that the threshold is the runtime of ϕ which is the upper bound runtime for all ψ sub-queries in the group. This threshold can be described by $\lceil \frac{\phi_r * \tau}{C} \rceil$ where τ is the execution threshold ratio, and C is a constant threshold step minimum enforced by Db2, currently set at 10. The Db2 thresholds are only created in 10-second intervals, so any threshold in-between is not possible. For example, the thresholds 10, 20, 30 are valid, but the thresholds 5, 15, 25 are not. This makes up for any queries that may terminate very closely after their ϕ upper-bound, but end up being terminated upon reaching the exact time. This, in essence, acts as a buffer to ensure those edge cases are mitigated, though not completely eliminated. This buffer though somewhat significant with fast-running queries does not have the same effect for slower running queries since 10-second intervals are much less significant. We thus have the motivation to find a method to cater to all queries, and also introduce a pruning rule as a result.

For each threshold, a *WORKLOAD* must be created in Db2, which can then have an associated *THRESHOLD* attached to it. This process is done for every g_{ijk}, thus creating a set of thresholds. The set of thresholds that are created can be formally described by $T = \{ \lceil \frac{\phi_r * \tau}{C} \rceil \mid \phi \, \epsilon \, g \, \epsilon \, G \}$ where G is the set of all groups of sub-queries described in Section 4.2. For every node in the cluster, the set of T thresholds is created, though this may seem excessive, it is a necessity due to the non-deterministic nature of Spark's distribution of sub-queries amongst the nodes. This means that there is no guarantee that any given ψ will end up on a specific node, and so to cover all ground, every threshold in T must be created on each node on the cluster. This is also done in a distributed fashion, but the time is negligible so we are able to only reap the benefits of the speedup without incurring any added cost.

Once the thresholds have been set up, each node will have a designated batch of ψ queries it must execute, and this is done concurrently using four executors, each utilizing a core from the four available. Similar to Φ query execution process described in Section 4.4, each sub-query is run a total of 3 times, to account for any noise the system may exhibit. It is possible that during any one of the three runs, the threshold limit is reached, and the Db2 manager terminates execution. This does not mean however that the remaining runs can be ignored since it is possible that later runs would result in faster execution. So if a run is terminated due to a threshold limit being reached, we would like to continue the subsequent runs in hope of finding one that performs better than the ϕ runtime.

Pruning The number of sub-queries generated per query is roughly a 200 times multiplier. For example, in a sample run with 99 TPC-DS queries, a total of 19845 sub-queries were generated, 2461 of which are ϕ optimizer chosen sub-queries, and the remaining 17384 are ψ randomly generated sub-queries. Each sub-query must also be executed three times to reduce noise, and so from just the 99 TPC-DS queries, we expect 59535 sub-query executions. Therefore, the ability to reduce any number of query executions is of great contribution to the speedup of the learning engine. We, therefore, introduce a pruning rule that aims to remove any runs that are deemed outliers and unable to contribute to templates.

We have mentioned that an upper bound already exists from the Φ sub-query execution, and a threshold slightly above that exists to terminate any slow-performing queries. Upon a sub-query hitting its threshold, it is terminated, but subsequent runs still continue. We make the argument, that any sub-queries executing on their first run and exceed their group's ϕ by 15% should not continue executing the second and third runs. The 15% threshold takes into account for any noise, and also the fact that the runtime is not the only determining factor in the ranking process. This reduces the number of sub-query executions by 1/3 for all sub-queries that meet this pruning rule. The main concern with pruning is potentially removing a ψ in the first run, that could potentially have a second or third run that outperforms ϕ and eventually contributes to a template. This is a scenario that is negated with a higher pruning threshold percentage, at the cost of a higher overall runtime. We have experimentally found that

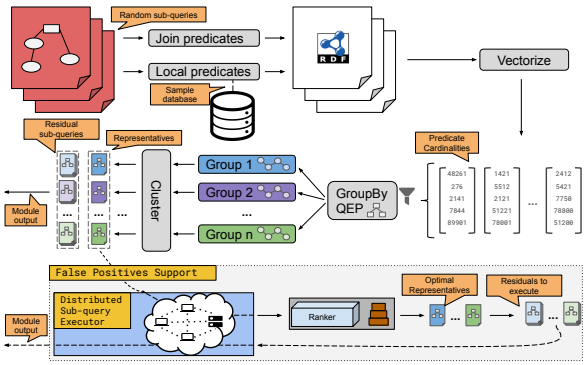


Figure 8: Random Sub-query Abrdiged Clustering Executor (RSACE) module.

15% finds that balance perfectly with minimal sacrifice on the number of templates discovered.

4.6 RSACE

Due to the nature of the sub-query generation, we observe a massive search space of random sub-queries that must be executed. In an effort to remedy the large search space, we have devised the Random sub-query Abridged Clustering Execution (RSACE) pruning module. The architecture of this component can be seen in Figure 8. This technique is only applied to the random sub-queries Ψ, since the runtimes of the optimizer sub-queries Φ are still required to create thresholds. This RSACE process is thus performed after the Φ execution, but before the Ψ execution, and is performed in several stages: predicate cardinality extraction, vectorization, QEP grouping, clustering, and finally a false positives support stage. We summarize the process of the RSACE module in Algorithm 2 and provide references below with specific line numbers.

In the predicate cardinality extraction stage, we extract all relevant information pertaining to predicates, including the values, tables, and cardinalities, all represented in a pseudo-RDF type format. These can be of two different types, local predicates, and join predicates. Since the cardinalities are of interest, we can obtain the actual join predicate cardinalities through the previously executed parent query (line 3). The query execution was the first step in the Learning process and therefore provides, not only the estimated but also the actual cardinalities of all the joins. Since queries subsume sub-queries, a sub-query's join will always be part of its parent, and thus the cardinality for any given sub-query will also be available. The local predicates were obtained during the sub-query generation phase, as with GALO, and are therefore easily parsed (line 4). Lastly, the *planID*, a table-independent hash value that defines the structure of the QEP, is parsed from the explain file (line 2). All queries with the same *planID* have the same execution plan and therefore since they are in the same ϕ group, also have the same predicate columns, though not necessarily the same predicate values. This helps group alike sub-queries that have the same structure but only differ in the predicate value.

During the vectorization step (line 5), the goal is to create a query to integer vector mapping for all Ψ sub-queries. The parsed information from the previous information is available in the pseudo-RDF format and allows for easy manipulation. The *vectorizePredicates* function extracts the sub-query's predicate,

input : The set of random subqueries to be executed Ψ
output: Cluster representatives $\mathfrak{R}$, or set of residuals Ψ'

1 **for** $\psi \in \Psi$ **do**
2 $\quad \psi$.planID = $parsePlanID(\psi$.QEP) ;
3 $\quad \psi$.predJoin = $extractFromActuals(\psi$.parentQuery) ;
4 $\quad \psi$.predLocal = $parseLocalPred(\psi$.QEP) ;
5 $\quad \psi$.predVector = $vectorizePredicates(\{\psi$.predJoin$\}$ $\cup$ $\{\psi$.predLocal$\}$) ;
6 **end**
7 $G = \Psi.groupBy(\psi \rightarrow \psi.planID)$;
8 Let $\mathfrak{R} = \{\}$ be the set of cluster representatives ;
9 **for** $g \in G$ **do**
10 $\quad \mathfrak{R} = \mathfrak{R} \cup \{hCluster(g)\}$;
11 **end**
12 **if** ***not*** $falsePositiveSupport$ **then**
13 $\quad$ return $\mathfrak{R}$;
14 **else**
15 $\quad distExec(\mathfrak{R})$;
16 $\quad$ Let $\Psi' = \{\}$ be set of optimal representative residuals ;
17 $\quad$ **for** $r \in \mathfrak{R}$ **do**
18 $\quad\quad$ **if** $r.execTime <$ $getOptimizerCounterpart(r).execTime$ **then**
19 $\quad\quad\quad \Psi' = \Psi' \cup flatten(r.\text{cluster})$;
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 $\quad$ return (Ψ') ;
23 **end**

Algorithm 2: Algorithm for RSACE Module.

hashes it, and saves it in a treemap; storing information in a sorted manner according to the natural ordering of the keys. This ensures that predicate cardinalities from different sub-queries are in the same indexed position of the vector. Each vector thus represents an ordered set of predicate cardinalities that can also be compared with other sub-queries with the same predicates.

A set of vector groups, G are created (line 7) to ensure that only sub-queries with the same predicates are compared. The grouping is based on the sub-query's QEP structure, such that sub-queries with identical QEPs are grouped together. This is achieved using the *planID* hash previously obtained. Since all Ψ sub-queries are derived from Φ sub-queries with guideline alterations, we observe a grouping of sub-queries with the identical guidelines, but not always derived from the same ϕ group. The process described so far can be summarized in Figure 9. In this example the actual cardinalities of join predicates **ws_item = i_item** and **ws_sold_date = d_date** are obtained from the parent query's QEP. The remaining predicates, **category = 'Jewelry'**, **category = 'Music'**, **d_date = '2016-01-02'**, and **d_date = '2016-12-01'**, have their estimated cardinalities calculated using the probe queries.

For each group in G, clusters of alike vectors are created (line 10), wherefrom within each, a single representative is selected. The sub-query representative should thus closely resemble all sub-queries within its cluster, since each is guaranteed to have the same execution plan, the same predicates, and now as we've established, very similar cardinalities. We can conclude that the runtime of the representative should be quite similar to the runtimes of the sub-queries within the cluster. The set of representatives $\mathfrak{R} \subseteq \Psi$ are the only needed sub-queries that require execution

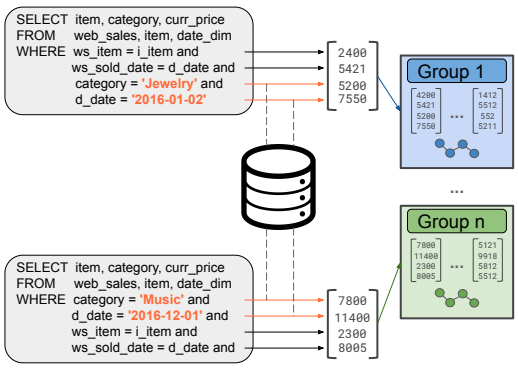


Figure 9: Random sub-query vectorization and grouping in RSACE.

since we can interpolate the runtimes of their corresponding clustered sub-queries. The clustering within each group G can be accomplished using various existing clustering techniques, but we observed that most required the number of clusters. We opted for the Density Based Spatial Clustering of Applications with Noise (DBSCAN)[15] algorithm, since it can find clusters of arbitrary size, and most importantly does not require an initial number of clusters. DBSCAN finds the number of clusters starting from the estimated density distribution of corresponding nodes. It requires the neighborhood of a point ϵ, and *MinPts* which is the minimum number of *density-connected* points required to form a cluster. Two points a and b are *density-connected* if there exist a set of points $\{ p_1, p_2, ..., p_n \}$ such that $p_1 = a$, $p_n = b$, and p_{i+1} is *directly density reachable* from p_i. Two points are *directly density reachable* from one another if they are within each other's ϵ neighborhoods. The remaining critical parts for the system are to select appropriate values for ϵ and *MinPts*. Since we observed relatively small groups and consequently clusters (varies with the number of random guidelines generated), a value of 0 for *MinPts* allows queries to be in their own cluster. This, in essence, means that outliers will be in the set $\mathfrak{R}$ and as a result will be executed. This is acceptable since it will reduce the false positives/negatives.

In most cases, the representatives $\mathfrak{R}$ adequately capture the runtimes but we must also consider times where they do not. The first of these is the false positive case, in which the ψ representative will **outperform** its ϕ counterpart, but is not the case for some or all ψ sub-queries in the corresponding cluster (residuals). The second is the false-negative case, in which the ψ representative will **under-perform** its ϕ counterpart, but some or all ψ sub-queries in the cluster outperform their ϕ counterparts. To remedy either case, the corresponding clusters could be re-run to correctly capture the runtimes of the sub-queries. This is however not a practical approach since this would encompass execution of all Ψ sub-queries, and defeat the purpose of the pruning. We must consider that, in general, ϕ tends to outperform ψ sub-queries, so quantitatively there will be less false positives than negatives. We can thus partially remedy these cases by executing all sub-queries in clusters of ψ representatives that outperformed their ϕ counterparts (line 15 - line 22). This ensures that all false positives are not miss-representing their cluster, and lead to faulty templates. The execution of the residuals is also done in a distributed fashion. False negatives are costly to address and thus we consider this approach a pruning rule, who's outcome depends on the quality of the clusters. This attainable through parameter tuning, or even different clustering algorithms.

5 EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of DistGALO in the following three categories:

(1) *Scalability*. We demonstrate the ability of DistGALO to more efficiently find problem patterns and handle larger databases previously considered unfeasible in GALO.
(2) *Distributed Performance*. We evaluate several cluster parameters and various partitioning strategies.
(3) *Pruning Effectiveness*. Finally, we present the effectiveness of the various pruning strategies employed, and what trade-offs they may present to the user.

Upholding the previous system GALO as a benchmark, we demonstrate the ability of our *Distributed Learning Engine* to more efficiently find templates. Furthermore, we display the system's ability to better handle larger data that was previously inconceivable to run in a reasonable amount of time. Finally, we showcase our pruning strategies and some of the benefits and inherent trade-offs they provide users. All experiments, with exceptions (noted), were conducted on the SOSCIP Cloud Computing Platform[3] using a cluster consisting of 9 nodes (including master). Each node consists of 4 Intel Core Processor (Haswell, no TSX, IBRS) 2.299 GHz Virtual CPUs, and 16GB of RAM. Experiments were conducted using the synthetic TPC-DS benchmark consisting of 25 tables, and 99 queries. All sub-queries considered in the evaluation consist of single, double, and triple joins, as was previously done in GALO and has shown to be most suitable for the synthetic data, but can otherwise be adjusted to any number of joins.

5.1 Distributed Learning Engine Evaluation

Exp-1: Revised Learning. One of the main motives for developing DistGALO was to address the lengthy learning time observed in GALO. Ultimately the goal was to reduce this learning time as much as possible, so as to permit larger data and more complex queries. In this experiment we compare the elapsed times of DistGALO and GALO from start (workload submission) to the end (template creation) of their respective (distributed) learning engines. The underlying learning process largely remains the same, mainly the decomposition of workload queries into their sub-query counterparts. Parameters across systems were kept similar including, three runs per sub-query for noise reduction, and 2/3/4way join sub-query decomposition.

The results are reported in Figure 10. The total elapsed time taken to learn the TPC-DS benchmark queries is shown in Figure 10a. The learning elapsed times on a per-query(Figure 10a) and a per-sub-query(Figure 10b) basis are reported as per GALO's *Learning Scalability and Effectiveness* experiment[11]. Though the focus previously was on the time complexity with respect to join-number, here we focus on the raw runtimes of each. First in Figure 10a we compare the total time taken to learn the workload from TPC-DS benchmark query set for a 1GB database. GALO would have sequentially taken 33, 319 hours versus the 10 hours it takes on DistGALO. On the previous system, learning on a 10GB TPC-DS query workload was not conceivable, but is now done in 20 hours.

System	Table Join Size	
	TPC-DS 1G	TPC-DS 10G
GALO (h)	33319.11	N/A
DistGALO (h)	10.41	20.24

(a) Learning time on all queries

System	Table Join Size		
	2-way	3-way	4-way
TPC-DS 1G			
GALO Time (s)	166218	315438	729948
DistGALO Time (s)	54.22	301.32	93.40
TPC-DS 10G			
GALO Time (s)	N/A	N/A	N/A
DistGALO Time (s)	148.82	438.66	295.97

(b) Learning time on a per-query basis

System	Table Join Size		
	2-way	3-way	4-way
TPC-DS 1G			
GALO Time (s)	27702	54090	88374
DistGALO Time (s)	1.23	2.97	1.27
TP-CDS 10G			
GALO Time (s)	N/A	N/A	N/A
DistGALO Time (s)	2.30	3.46	2.61

(c) Learning time on a per-subuery basis

Figure 10: DistGALO's learning elapsed time (seconds) versus DistGALO using TPC-DS benchmark queries.

Next, we consider the time it takes to learn a single query(Figure 10b) we observe a drastic drop from 166218 to 54.2 seconds for 2-way join sub-queries, and similar speedups for 3-way and 4-way join sub-queries.

Analyzing the runtimes on a per-query basis however, doesn't accurately depict the overall speedup achieved through DistGALO. Different queries produce varying amounts of sub-queries at the varying levels of join-numbers, thus we only observe an average. Let us consider the amount of time taken to learn a single sub-query (Figure 10c). GALO was previously reporting 27702(7.6 hours), 54090(15.0 hours), 88374(24.5 hours) for 2-way, 3-way, and 4-way join sub-queries respectively. All the runtimes were reduced to a sub-5-second range using the current Distributed Learning Engine.

The drastic difference can thus be attributed to the various partitioning strategies, and other optimizations like multi-executor concurrency execution, selective thresholds, and others mentioned throughout this work. The results are devoid of any pruning rules to maximize template creation, resulting in 108 templates found, in comparison to the 98 found in GALO over the TPC-DS 99 queries. The discrepancy we attribute to the randomness of the generated sub-queries since no fundamental process was changed in how sub-queries are generated, ranked, and used to create templates. The DistGALO average performance improvement over the problematic sub-queries is 35% in comparison to the GALO reported 37%. This we too attribute to the random nature of the system when creating sub-queries as was done for both GALO and DistGALO.

Exp-2: Database Scalability. Next we consider how DistGALO is able to scale in accordance to an increased database size

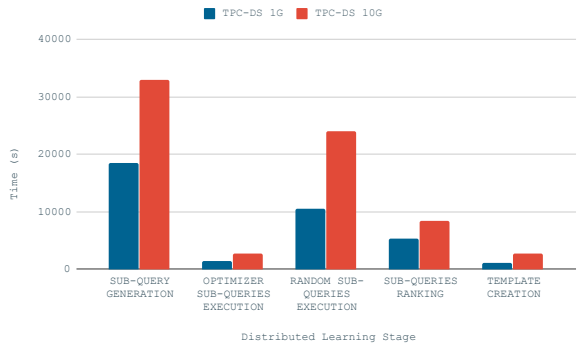


Figure 11: Distributed Learning with 1G and 10G TPC-DS databases.

of 10GB. This was an inconceivable task with the previous GALO system, taking upwards of 48 hours before it was terminated by force. We, therefore, are able to showcase a feature beyond just improvement, but rather a whole new ability to tackle large data. This we hope is one step closer to bringing the system to meet modern-day data requirements, with some exhibiting sizes beyond the petabyte range. However, due to limited hardware available, we have opted for a 10x approach, showcasing the ability to handle a database size of 10GB versus the 1GB TPC-DS benchmark.

We only consider the Distributed Learning Engine since the Matching Engine remains largely unmodified, and was proven to scale well[11]. We thus split up the Learning Engine process into several smaller stages: *sub-query generation*, *optimizer sub-query execution*, *random sub-query execution*, *sub-query ranking*, and *template creation*. We further analyze the runtime of each of these and discuss the observations made.

Figure 11 displays the various stages of the Distributed Learning Engine in a 1GB versus a 10GB environment over the TPC-DS benchmark. The experiment demonstrates how DistGALO is effectively able to handle a database scaling of 10X. Our results reveal a learning runtime increase from approximately 10.4 hours for TPC-DS 1G, to 20.2 hours for TPC-DS 10G. This 2X scalability experiment shows how effective the system handles an increase in data size, which is much more in line with current-day demands. Two stages in the learning process are worth mentioning, mainly the *sub-query generation* and *random sub-query execution*, since they take a considerably longer time than the remaining. The latter proved to be a major bottleneck in GALO and therefore was the primary area of focus for optimization in DistGALO. This we believe was successfully achieved since it no longer stands to be the bottleneck of the learning process. Consequently, however, we observe a new bottleneck, albeit smaller than previous, emerge. In both 1G and 10G TPC-DS datasets, the *sub-query generation* makes up for 50% of the total learning runtime in both cases. It currently stands to be the main time-sensitive obstacle largely attributed to the probing queries used to obtain local predicate ranges. This step though already distributed will likely be the main focus of our future work, in ensuring an even more effective learning process.

5.2 Distributed Performance

Exp-3: Executor Impact. Despite distributing the work amongst nodes, we are able to leverage the parallelism within nodes in

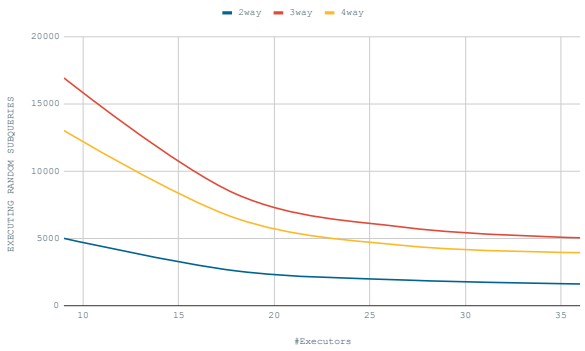


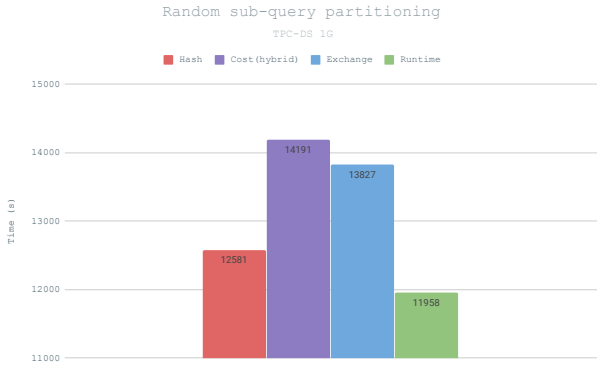
Figure 12: Random sub-query execution with different number executors.

order to further optimize. This is achieved by utilizing distributed workers, or *executors*, in order to make use of all available cores on each node. Each executor, in the context of DistGALO, is responsible for executing a single sub-query, ranking all runs, and storing the results in the distributed file system. We experiment with various executors, starting at 9, the number of nodes, up to 36, the total number of virtual CPUs on the cluster. The results are shown in Figure 12.

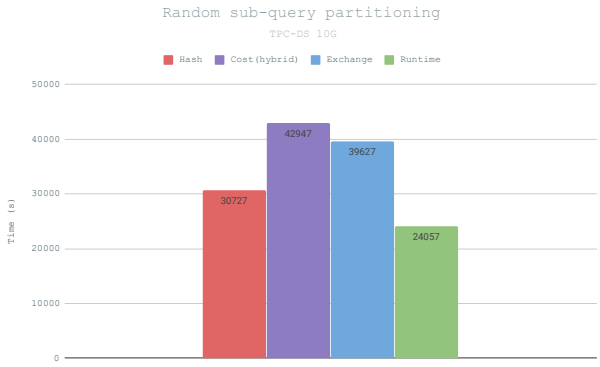
The results are quite intuitive since with 9 executors, we are only able to achieve serial execution on each node using 1 VCPU, and the remaining 3 on each node are left idle. This highly underutilizes resources and shows with a higher overall execution time. We then observe a gradual decrease as more executors are added until we reach 36, the total number of cores available. These results though not quite surprising have great importance. What we can deduct is that regardless of how the system scales, be it vertical or horizontal, so long as the number of executors is increased, there will be a definite gain. This gives great flexibility to users in terms of scalability, since some have more powerful nodes but lesser in number, whilst others may have less powerful nodes but in greater quantity. In either case, DistGALO will ensure that all CPUs across the cluster are fully utilized.

Exp-4: Random Sub-query Execution Partitioning. In order to minimize the amount of skew exhibited by the cluster, nodes must be balanced as evenly as possible. Skew can be greatly reduced by minimizing communication cost between nodes, and partitioning allows that fine-grained control. Partitioning allows DistGALO to group certain sub-queries together, selected by the partitioning strategy. We previously discussed the different strategies (Section 4.5) employed by the DLC: Hash, Cost Estimate (Hybrid), Exchange, and Runtime Partitioners. We observe the effect these various partitioners have on the random sub-query execution in Figure 13. The experiments were conducted on both TPC-DS 1G and TPC-DS 10G to get a sense of the scalability of the different partitioners.

The Runtime Partitioner is consistently the lowest runtime, with an overall learning execution time of 11, 958 and 24, 057 seconds for TPC-DS 1G (Figure 13a) and TPC-DS 10G (Figure 13b)respectively. This is mainly attributed to the fact that the random sub-query runtimes are bounded by the thresholds imposed by DistGALO. These thresholds are derived from the optimizer sub-query runtimes, which is the same data that is used to partition the random sub-queries. In essence, each partition's value



(a) Partitioners on TPC-DS 1G.



(b) Partitioners on TPC-DS 10G.

Figure 13: Random sub-query execution using Hash, Cost Estimate (Hybrid), and Runtime Partitioners.

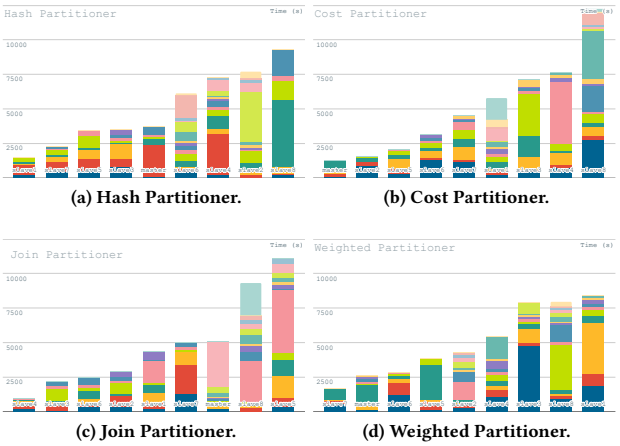
denotes the overall runtime for that group of sub-queries within the partition.

The Hash Partitioner is, in essence, distributing sub-queries as evenly as possible, without any consideration of the internal structure or cost. The Cost (hybrid) Partitioner, on the other hand, takes into account the estimates calculated by the Db2 optimizer and so each partition's value will be a timeron value corresponding to the estimated execution time for that group. This strategy leverages the heuristics of the optimizer thus exhibiting some form of intelligence over the hash partitioner, and should have better overall performance. As indicated in our results, this is however not the case. The hash partitioner outperforms the Cost Partitioner, invalidating the hypothesis formerly stated. Further analysis shows that the sub-queries typically fall into one of two extremes, one where they execute very quickly, and another where they hit the threshold, terminating execution. In the case of TPC-DS 1G in Figure 13a, the extremes are much more emphasized, and so a strategy of evenly distributing the workload, like in the Hash Partitioner, is actually quite effective. This is further compounded by the instances of inaccurate costs, though this is much harder to quantify. In the case of the TPC-DS 10G, the extremes are lesser, thus making the Hash Partitioner, less effective, though still enough to outperform the heuristic-based approach. The gap between the Hash and Cost (hybrid) partitioned has lessened, and we hypothesize that with a larger dataset,

the Cost (hybrid) approach will eventually outperform the Hash Partitioner.

The last partitioner left to be mentioned is the Weighted Partitioner, which is also outperformed by the Hash Partitioner in both instances. Since the exchange rate is derived from the cost, and the estimated random sub-query runtime is also estimated based on cost, it ultimately suffers the same fate as the Cost (hybrid) Partitioner. The noteworthy observation is that by using the derived exchange rate, this approach is able to provide a more accurate heuristic than the timerons in the Cost (hybrid) approach. This doesn't incur any lengthy computation to calculate and is overall the preferred method between the two partitioners. Even though it inherits the same unfortunate characteristic as the Cost (hybrid) partitioner, we also hypothesize that it also inherits the potential to scale with data and eventually would outperform the Hash Partitioner.

Exp-5: Sub-query generation partitioning strategies.



(a) Hash Partitioner. **(b) Cost Partitioner.**

(c) Join Partitioner. **(d) Weighted Partitioner.**

Figure 14: Sub-query generation Partitioners.

The sub-query generation process utilizes one of four available partitioners: the Hash, Cost, Join, and Weighted Partitioner. Figure 14 breaks down the TPC-DS 10G 3-way sub-query generation execution of each node into the different queries, denoted with different colors, and also the runtime of each, denoted by the length of each color. These are also sorted into ascending order, from the fastest executing node to the slowest, the latter dictating the overall runtime. The steeper the slope, or curve, of the invisible line running atop the bars in the graph, the greater the skew exhibited. The total runtime is described by the longest-running (rightmost) node in each graph.

The worst performing is the Cost Partitioner in Figure 14b with a total runtime of 12, 265 seconds. The cost provides insight into the complexity of the query, but fails to factor in any information regarding the decomposition process of generating sub-queries. It may be so that the original query is quite complex due to some expanding joins, but the individual tables themselves are quite simple and quick to probe, thus having a relatively speedy generation step. This approach is not able to accurately capture the characteristics of a long-executing query. This is evident when analyzing the longest-running node on the right, which has grouped several such long-executing queries, though somewhat successfully grouping the other nodes. The Join Partitioner in Figure 14c is similar in nature, but only relies on the single heuristic which is that a query that generates more

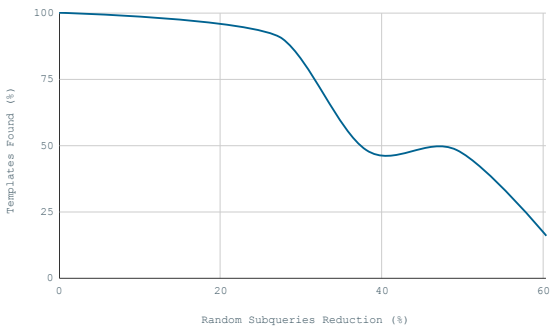


Figure 15: RSACE random sub-query reduction and its effect on number of templates found.

sub-queries typically takes longer to execute since there are more probes to be made. It has a marginally better performance than the Cost Partitioner, with an overall runtime of 11, 043 seconds. By simply relying on this heuristic alone and ignoring any other information regarding complexity it also wrongly groups queries which may produce many sub-queries, but are quick to probe. The rightmost two nodes have several long-executing queries that should have never been grouped on the same node and consequently introduce a considerable amount of skew.

The Hash Partitioner in Figure 14a is the second-best performing, with a total runtime of 9, 303 seconds. This partitioner simply distributes queries as evenly as possible based on count. This seems to be a relatively effective approach and seems to perform quite well with this workload, but some of its success can be attributed to chance. With a different workload, it may group several long-running queries together to introduce considerable skew. The best performing is the Weighted Partitioner in Figure 14d, with a total runtime of 8, 403 seconds. This approach captures the essence of both the Cost and Join Partitioners by combining them and associating a weight to each. The weights used in this evaluation given even distribution, with each set to 0.5. This has experimentally shown to be the most effective but may require modification on different workloads. By capturing the complexity of the query and also taking into account the number of to-be generated sub-queries, the partitioner is accurately able to distinguish the long-running queries and put them on separate nodes. This can also visually be confirmed with three right-most nodes in Figure 14d, which have been evenly given the three longest-running queries of the workload. In further development of this work, we hope to automatically adjust weights with different workloads, based on some quickly scanned heuristics of the queries.

5.3 Pruning Effectiveness

Exp-6: RSACE Effectiveness. DistGALO aims to provide more versatile and flexible parameters for users to adjust. The RSACE module prunes sub-queries of similar structure and predicates, so that two very similar sub-queries need not be executed twice. The pruning process relies on clustering groups of sub-queries and selecting representatives $\mathfrak{R}$ to be executed. The better the representative can summarize members of its group, the more can be inferred about those members, and therefore the smaller the search space. Consider that any clustering algorithm can be chosen, but in our experiments, we opted for the DBSCAN clustering algorithm [16] since it is able to find clusters of arbitrary

sizes, and does not require an initial cluster size. DBSCAN's aggressiveness can be adjusted with a larger neighborhood range, controlled by the ϵ parameter. By increasing the ϵ value, we more aggressively group sub-queries, thus resulting in fewer representatives and therefore less random sub-queries that must be executed. The aggressiveness of the pruning will have some effect on the accuracy, or the number of templates discovered in the context of DistGALO. With a more aggressive approach intuition dictates a lower runtime, with a compromise on the number of templates discovered. The unknown is how expensive that trade-off is, and at what intervals. The effectiveness of the RSACE pruning strategy with varying aggressive parameters can be seen in Figure 15.

The worst-case result for the experiment would be a $-log$ effect, where minor pruning adjustments cause a drastic loss in accuracy. Our results show an almost inverse effect for the range of (0%, 27%] and (38%, 48%] of pruned random sub-queries. The former indicates that an ideal selection of aggressiveness is at an ϵ value that prunes 27% of random sub-queries, while still identifying 90% of the templates that would otherwise be discovered without RSACE. Beyond this range, there is a large loss in the number of templates found with only 48% being discovered with 38% of sub-queries pruned. Following that we exhibit an odd behavior in which higher pruning results in more templates being discovered. This can be attributed to the possibility of a cluster restructure causing different representatives to be selected, marginally more representative of the groups, thus resulting in more templates. This we consider an anomaly as the trend continues past 50% pruning. We conclude that the RSACE module is quite effective as it can provide a 27% speedup for a 10% loss in accuracy.

6 RELATED WORK

The classical cost-based method for selecting an efficient query plan dates back to the predecessor of all modern-day relational databases, System R[6]. This relational approach to database management pioneered the cost-based optimizer methodology of obtaining a low-cost means for query execution. The optimizer's cost-based metric depends on disk page accesses, including CPU instructions, with an effort to minimize the number of pages being fetched from secondary storage. This early work applied a bottom-up dynamic programming plan enumeration technique to efficiently create a sub-set of plans from a massive search space. Since then, many advancements have been made in query optimization techniques, with modern-day systems regarding a high demand since the Big Data Revolution[31].

One approach to advancing the optimization process is to devise new strategies for pruning the join order search space. Finding the optimal access plan is an NP-hard problem, and thus the classic Dynamic Programming approaches switch to heuristic or randomized methods to resolve complex, high join order, queries. Methods have been devised so that when a query becomes too complex to be optimized accurately, the join graph is reduced to a simpler one until it becomes tractable within a given time budget[29]. Alternative methods introduce a top-down join enumeration algorithm, accompanied by various branch-and-bound pruning techniques[16].

Another heavily researched approach has been to improve the accuracy of cardinality estimation, as it is a chief determinant of the overall cost of a plan. The traditional and dominant histogram-based approach independently calculates selectivities of local predicates, while disregarding statistical correlations when calculating the net selectivity. This results in inaccurate cardinality estimates with more complex and often real-world data sets[27][23], since they don't fit the assumptions of uniformity, independence, and inclusion that optimizers make. Several sampling methods[18][19][25] attempt to circumvent this issue since they are better able to capture data correlation. A recent approach uses index-based join sampling, where more accurate cardinality estimates are derived from existing index structures and sampling[24]. This technique takes advantage of the recent advancements of in-memory databases and leverages sampling with a designated operator that utilizes already-existing index structures.

Recently Machine Learning has been a catalyst in many fields within research and development, and query optimization has not been an exception[35]. Neural Networks have been used to accurately estimate the selectivity of queries over highly skewed or correlated data[26]. Other Deep Learning techniques[22][28][30] structure the join ordering as a reinforcement learning problem to obtain the query plans, while others apply supervised learning to solve cardinality estimation in isolation[21]. Machine Learning has proved an effective enough method and could revolutionize the database optimizers of the future.

The discussed methods thus far have been an integral component or modification of the cost-based optimization stage of the query compiler. The matter of fact is that though there has been considerable improvement, the optimizer will not always be able to pick the optimal access plan, and domain experts will consequently always be needed to troubleshoot such cases. It is the troubleshooting stage for which this work makes an effort to automate, not the query optimization itself. This third-tier optimization step, we term the *plan re-optimization* stage, in which the chosen optimizer plan is rewritten such that subsequent executions suggest a more optimal plan to the optimizer. The first undertaking, OptImatch[12][13] aimed to aid domain experts by letting them build problem patterns, store, and later retrieve to automatically apply to sub-optimal queries. This proved to be a useful tool but still required some manual input in order to populate the knowledge base. To remedy this, we devised GALO[11][10], a system capable of automating the entire process from end to end, without any domain expert intervention. It was capable of automatically discovering and saving problem patterns into a knowledge base, later applied as rewrites to non-optimal access plans. It still filled the requirements of OptImatch in that it acted as a query re-optimization tool, applied post query compiler evaluation. A similar strategy of query re-optimization[36] aims to automatically provide refined, more accurate, sampled cardinalities to sub-optimal queries. This allows the optimizer to take the updated cardinalities and make a new, more informed decisions, resulting in more optimal access plans. This process would be repeated until the refined cardinalities being fed no longer lead to a different access plan. The difference between the latter mentioned approach and this work is that DistGALO relies on real run-time statistics of sub-queries, and is thus able to make objective statements that are not influenced by optimizer heuristics.

7 CONCLUSION

In this work we improve upon the well-received GALO system with DistGALO by revamping the Learning Engine into a Distributed Learning Engine. We utilize modern-day scalable

technologies[31] like Apache Spark and Hadoop Distributed File System, to effectively and efficiently populate its knowledge base. DistGALO is able to leverage the availability of cloud machines to allow on-demand horizontal scalability to fit the workload domain requirements. Using various partitioning strategies, we effectively minimize the skew throughout nodes. We also introduce pruning strategies that decrease the redundancy throughout executions, whilst maintaining its effectiveness in learning problematic patterns. Despite the numerous structural changes, the technique remains the same, and DistGALO can still be regarded as third, *plan re-optimization*, stage of query optimization. New bottlenecks have emerged, mainly the sub-query generation, as we find the predicate probing queries to be expensive. New methods for generating sub-queries and translating them to RDF could be explored, potentially accomplishing it directly from the QGM without any intermediate parsing. Automated scaling could also trivially be applied so that heuristics and metadata could be obtained from query workloads, to automatically adjust the cluster size as needed. The growing popularity of containerization could aid in the dynamic ability to load balance and scale DistGALO. The system has proved to be very effective and continues to evolve as the technologies it relies on also evolve.

REFERENCES

[1] [n. d.]. OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes. https://www.openshift.com/. ([n. d.]). (Accessed on 10/21/2019).

[2] [n. d.]. Production-Grade Container Orchestration - Kubernetes. https://kubernetes.io/. ([n. d.]). (Accessed on 10/21/2019).

[3] [n. d.]. SOSCIP | Smart Computing for Innovation. https://www.soscip.org/. ([n. d.]). (Accessed on 10/21/2019).

[4] 2018. IBM InfoSphere Optim Query Workload Tuner. https://www.ibm.com/support/knowledgecenter. (2018). https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.idm.tools.doc/doc/c0057033.html

[5] Ritu Agarwal and Vasant Dhar. 2014. Big data, data science, and analytics: The opportunity and challenge for IS research. (2014).

[6] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)* 1, 2 (1976), 97–137.

[7] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. 2009. Interactive plan hints for query optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 1043–1046.

[8] Nicolas Bruno, Surajit Chaudhuri, and Ravi Ramamurthy. 2009. Power hints for query optimization. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 469–480.

[9] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. 2004. Automatic SQL tuning in oracle 10g. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 1098–1109.

[10] Guilherme Damasio, Spencer Bryson, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. GALO: guided automated learning for re-optimization. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1778–1781.

[11] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alexandar Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. Guided Automated Learning for query workload re-Optimization. *arXiv preprint arXiv:1901.02049* (2019).

[12] Guilherme Damasio, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. 2016. OptImatch: Semantic web system for query problem determination. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1334–1337.

[13] Guilherme Damasio, Piotr Mierzejewski, Jaroslaw Szlichta, and Calisto Zuzarte. 2016. Query Performance Problem Determination with Knowledge Base in Semantic Web System OptImatch.. In *EDBT*. 515–526.

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[16] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. 2012. Effective and robust pruning for top-down join enumeration algorithms. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 414–425.

[17] Jarek Gryz, Qiong Wang, Xiaoyan Qian, and Calisto Zuzarte. 2008. SQL queries with CASE expressions. In *International Symposium on Methodologies for Intelligent Systems*. Springer, 351–360.

[18] Peter J Haas, Jeffrey F Naughton, S Seshadri, and Arun N Swami. 1996. Selectivity and cost estimation for joins based on random sampling. *J. Comput. System Sci.* 52, 3 (1996), 550–569.

[19] Peter J Haas and Arun N Swami. 1995. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE, 522–531.

[20] Matthias Jarke and Jurgen Koch. 1984. Query optimization in database systems. *ACM Computing surveys (CsUR)* 16, 2 (1984), 111–152.

[21] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[22] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[24] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling.. In *CIDR*.

[25] Richard J Lipton, Jeffrey F Naughton, and Donovan A Schneider. 1990. *Practical selectivity estimation through adaptive sampling*. Vol. 19. ACM.

[26] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. 2015. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 53–59.

[27] Guy Lohman. 2014. Is query optimization a âĂIJsolvedâĂİ problem. In *Proc. Workshop on Database Query Optimization*, Vol. 13. Oregon Graduate Center Comp. Sci. Tech. Rep.

[28] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, 3.

[29] Thomas Neumann. 2009. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 403–414.

[30] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning state representations for query optimization with deep reinforcement learning. *arXiv preprint arXiv:1803.08604* (2018).

[31] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. 2018. Big Data technologies: A survey. *Journal of King Saud University-Computer and Information Sciences* 30, 4 (2018), 431–448.

[32] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *ACM SIGMOD Record*, Vol. 29. ACM, 249–260.

[33] David Simmen, Eugene Shekita, and Timothy Malkemus. 1996. Fundamental techniques for order optimization. In *ACM SIGMOD Record*, Vol. 25. ACM, 57–67.

[34] Uthayasankar Sivarajah, Muhammad Mustafa Kamal, Zahir Irani, and Vishanth Weerakkody. 2017. Critical analysis of Big Data challenges and analytical methods. *Journal of Business Research* 70 (2017), 263–286.

[35] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database meets deep learning: Challenges and opportunities. *ACM SIGMOD Record* 45, 2 (2016), 17–22.

[36] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1721–1736.

[37] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[38] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.