gCBO: A Cost-based Optimizer for Graph Databases

Linglin Yang
Peking University
Beijing, China
linglinyang@stu.pku.edu.cn

Lei Yang
Peking University
Beijing, China
yang_lei@pku.edu.cn

Yue Pang
Peking University
Beijing, China
michelle.py@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

ABSTRACT

Query optimization is an especially challenging problem in graph databases due to its wide plan space and the difficulty in gathering statistics. In this demonstration, we propose a new cost-based query optimizer called gCBO for graph databases and implement it in a specific graph database (i.e., gStore). To tackle the aforementioned challenges, gCBO employs a hybrid plan enumerator based on dynamic programming, cost models that capture the characteristics of different types of joins, and a sampling-based cardinality estimation strategy that gathers the necessary statistics on-the-fly. What is more, to absorb the experience of users, we build an interactive component for gCBO, which allows users to receive the optimized execution plans with detailed information and generate their own plans for execution.

CCS CONCEPTS

• **Information systems → Query optimization**; *Database management system engines*; *Database query processing*.

KEYWORDS

Graph database; query optimization; SPARQL query

ACM Reference Format:
Linglin Yang, Lei Yang, Yue Pang, and Lei Zou. 2022. gCBO: A Cost-based Optimizer for Graph Databases. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3511808.3557197

1 INTRODUCTION

Query optimization is one of the most challenging problems in database systems. A good optimizer can speed up the execution time by up to orders of magnitude through various optimization techniques [4]. Query optimization in the recently popular graph databases is even more crucial, for graph queries often involve a large number of joins and thus require more sophisticated optimization techniques to achieve high performance on execution. There are three major difficulties in the query optimization for graph databases:

(1) **Wide Plan Space.** Queries on graphs are usually complicated with many joins, so the space of query plans is wide when considering the join orders. This makes greedy strategies for join order selection employed in graph databases (e.g., gStore [8]) especially vulnerable to getting stuck in local optima. Moreover, this problem is exacerbated when considering storage implementations (e.g., index selection) and query types (e.g., limit-k queries).

(2) **Difficulty in gathering statistics.** Many relational databases rely on some forms of statistics (e.g., histograms) for query optimization. However, these techniques often fail in graph databases for two reasons. First, such statistics are not well-defined on graphs as on relational tables. Second, such statistics are expensive to maintain due to the large time and space overhead. For example, Graphflow's query optimizer [5] is effective on graphs with few labels based on its statistics called catalogue. However, for real-world graphs with only dozens of labels, it can easily take more than one day to build such a catalogue [3].

(3) **Lack of user interaction.** Sometimes, the experience of users can help generate better execution plans or correct obvious mistakes of the database optimizer. However, existing graph database systems hardly provide such interactive interfaces for users to utilize their feedback during query execution.

Our Methods and Contributions. To overcome these difficulties, we absorb experience from relational database query optimization to design a **C**ost-**B**ased **O**ptimizer for **g**raph database systems named **gCBO**, and implement it in gStore[1]. Specifically:

- We design a plan enumerator based on dynamic programming, which effectively enumerates query plans combining worst-case-optimal (WCO) joins and binary joins, and propose distinct cost models for queries with or without `LIMIT` clauses. The combination of our plan enumerator and cost model enables our optimizer to find a near-optimal plan in a relatively short time.
- We propose a sampling-based cardinality estimation strategy, which obtains the necessary statistics on-the-fly with low overhead. Our method ensures that we predict the cardinality efficiently without computing or storing any redundant statistics.
- We build an interactive component for visualizing query optimization[2], which displays the tree structure of the optimized query plan, the execution time of each operation, the real cardinalities and our estimated ones. Moreover, after investigating the query plan generated by the optimizer, users can revise or rewrite the query plans for execution via the interactive interface.

2 DEFINITION AND NOTATIONS

In this work, we focus on graph database query optimization. Our method is based on RDF data graphs and SPARQL queries. However,

CIKM '22, October 17–21, 2022, Atlanta, GA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9236-5/22/10…$15.00
https://doi.org/10.1145/3511808.3557197

[1]https://github.com/pkumod/gStore.
[2]https://github.com/pkumod/gStore_plan_presentation.

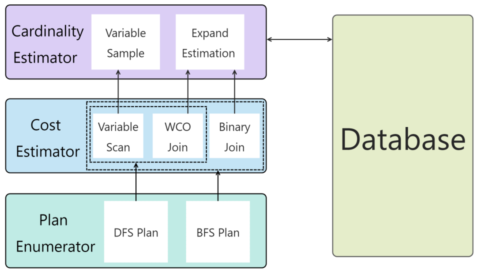


Figure 1: Architecture of gCBO

our optimizer fits other graph data models such as property graphs as well.

An RDF data graph consists of a set of triples $\langle s\ p\ o \rangle$ representing relations between entities or attributes of certain entities, where s, p and o are called the subject, predicate and object, respectively.

SPARQL [2] is the standard query language for RDF data graphs. A simple SPARQL query consists of a header (i.e., the `SELECT` clause, which specifies the projection variables in the final results) and a body (i.e., the `WHERE` clause, which specifies the graph patterns to be matched). In this demonstration, we only consider `SELECT` queries with a connected basic graph pattern. A basic graph pattern comprises triple patterns in the form of $\langle s\ p\ o \rangle$, where the subject, predicate and object may either be a constant (e.g., $\langle p \rangle$) or a variable (e.g., $?x$). It is connected when the triple patterns form a connected graph. For convenience, we only introduce our methods on queries with constant predicates. However, our optimizer can handle predicate variables as well. At the end of a query, users can constrain the number of results by an upper bound using a `LIMIT` clause. In this demonstration, we call queries with and without `LIMIT` clauses *limit-k* queries and *normal* queries, respectively.

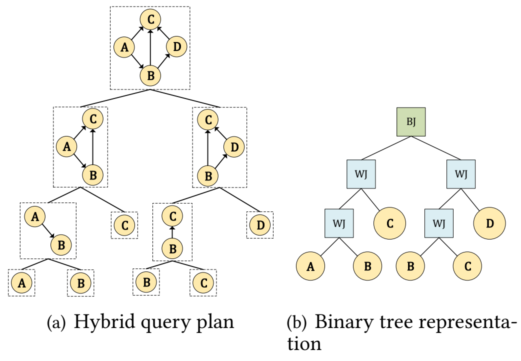
An example query is shown below. We can classify the triple patterns into two types: one is called *constant constraints*, which describe the attributes of variables on the subject or object position (like $\langle ?x\ \langle p_1 \rangle\ \langle o \rangle \rangle$, only one variable in each triple); the other is called *variable joins*, which link two variables by a constant predicate (like $\langle ?x\ \langle p_2 \rangle\ ?y \rangle$). Our goal is to generate near-optimal execution plans for such queries.

```
SELECT ?x ?y ··· WHERE{
    ?x    ⟨p1⟩    ⟨o⟩.      # constant constraint
    ······
    ?x    ⟨p2⟩    ?y.       # variable join
}   (LIMIT k)               # LIMIT clause (optional)
```

3 COST-BASED OPTIMIZER

3.1 System Overview

Figure 1 shows the architecture of gCBO. It consists of three main components: (1) plan enumerator, (2) cost estimator and (3) cardinality estimator. The plan enumerator enumerates possible execution plans. Then the cost estimator invokes the cardinality estimator to estimate the execution cost of the current plan. Finally, the optimizer selects the plan with the least cost to execute.



(a) Hybrid query plan (b) Binary tree representation

Figure 2: An example of a hybrid query plan and its corresponding binary tree representation.

3.2 Plan Tree Representation

Before introducing how plans are enumerated, we first specify the plan space that we consider. Our execution plan may comprise the following two types of joins:

- Binary join, which joins two bags of subquery results to get the results of a larger set of query variables, similar to natural join in RDBMS. Binary joins can be implemented by well-studied methods, such as hash-join or merge-join.
- WCO join [6, 7], which extends a vertex and processes all edges that link the new vertex and the previous subquery graph at a time.

Either of these two join methods could be more efficient than the other in different scenarios. For example, recent studies have empirically shown that WCO joins perform better on highly cyclic queries, while binary joins are more efficient on cycle-free queries. [1, 6]. We therefore consider hybrid plans that combine these two join methods. Figure 2(a) shows an example of such a hybrid plan.

gCBO uses a binary tree structure to represent the query plan. For instance, the execution plan of Figure 2(a) is shown as a tree in Figure 2(b). Each *leaf node* of the binary tree is one-to-one mapped to a query variable which represents a variable scan operation. For example, for the triple pattern $\langle ?x\ \langle p \rangle\ \langle o \rangle \rangle$, there exists a leaf node $?x$ which scans the database and sets those subjects linked with the predicate $\langle p \rangle$ and object $\langle o \rangle$ as the *candidates* of the variable $?x$. Moreover, if the variable occurs in multiple triple patterns with constants, these candidate lists are intersected to produce the final candidates of that variable in one scan operation.

Internal nodes of the plan tree represent joins between the results of the two subtrees rooted at the join node. For WCO join nodes, we place the extended variable as the right child. For binary join nodes, we require that their two child subtrees have at least one common leaf node.

3.3 Dynamic Programming Plan Enumerator

To handle the exponential plan space, we designed a dynamic programming plan enumerator. The pseudocode of our enumerator is shown in Algorithm 1.

The enumerator receives a query Q whose variable set is denoted as V_Q and outputs an execution plan denoted by T_Q expressed in the binary tree form. After performing variable scans on all the query nodes (Line 1), the enumerator expands the subquery step

Algorithm 1: Dynamic Programming Plan Enumerator

```
Input: SPARQL query $Q(V_Q)$
Output: Query plan $T(Q)$ of the input query $Q$
1  Perform variable scans on all query nodes ;
2  for $k$ from 2 to $|V_Q|$ do
3      foreach $V_{k-1} \subseteq V_Q$, s.t. $|V_{k-1}| = k-1$ do
4          bestPlan ← BestPlan($V_{k-1}$) ;
5          foreach $v$ in Neighbor($V_{k-1}$) do
6              P ← WCOJoin(bestPlan, $v$) ;
7              insert P to PlanCache ;
8      if $k \geq 5$ and Type($Q$) = NormalQuery then
9          foreach $V_k \subseteq V_Q$, s.t. $|V_k| = k$ do
10             for $V_{k_1}, V_{k_2} \subseteq V_k$, s.t. $V = V_{k_1} \cup V_{k_2}$ do
11                 P ← BinaryJoin(BestPlan($V_{k_1}$), BestPlan($V_{k_2}$)) ;
12                 if cost(P) < BestPlan($V_k$) then
13                     insert P to PlanCache ;
14 return BestPlan($V_Q$)
```

by step by expanding the set of query nodes. There are two modes of extension corresponding to the two types of joins: WCO join, which adds a neighbor of any node currently in the set to the subquery (Lines 3-7); and binary join (Lines 8-13). Notice that we only consider binary join when the set of query nodes is large enough (i.e., when $|V_Q| \geqslant 5$) since binary joins are equivalent to WCO joins if $|V_Q| < 5$.

3.4 Cost Estimator

In order to compare the quality of different query plans and select a near-optimal plan for the executor, we need to estimate their execution costs (Lines 6 and 11 of Algorithm 1), including the computation cost and data transfer latency, and choose the plan with the lowest cost (the invocations of `BestPlan` in Algorithm 1).

As mentioned in Section 3.2, an execution plan T is composed of join operations and variable scans. Because variable scans are completed at the plan enumeration stage (Line 1 in Algorithm 1), the cost of the plan $T(Q)$ can be expressed as the sum of the costs of its join operations:

$$cost(T(Q)) = \sum_{J \in T(Q)} cost(J) \tag{1}$$

(1) **Cost of WCO joins.** For WCO join operation, our executor checks each of the edges adjacent to the newly added node in the query graph for each result from the last step. For each edge, according to the subject-predicate pair (or the object-predicate pair, depending on the edge direction), the executor retrieves the corresponding set of objects (or subjects) from the database. Finally, these retrieved sets and the candidate set of the newly added query node are intersected to get the result set of the current step. Therefore, we can estimate the cost of WCO join with the following formula:

$$\begin{gathered} cost(\text{WCOJoin}(\{v_1 \cdots, v_{k-1}\}, v_k)) = \\ card(\{v_1, \cdots, v_{k-1}\}) \times \min_{\substack{\langle v_i\ p\ v_k\rangle in\ Q \\ or\ \langle v_k\ p\ v_i\rangle in\ Q}} average\_size(v_i, p) \end{gathered} \tag{2}$$

This formula emulates the cost of checking the aforementioned edges regarding the newly added vertex v_k. For each edge, according to the procedure, we need to scan the set of matching objects (or subjects, depending on the edge direction) whose estimated size is captured by $average\_size(v_i, p)$. This estimation can be done by averaging over samples during cardinality estimation when $k = 2$. Multiplying the minimum value of these estimates by the previous cardinality estimate $card(\{v_1, \cdots, v_{k-1}\})$, we get an upper bound on the join results, which is then used to approximate the cost of the WCO join.

(2) **Cost of binary joins.** The binary join is implemented as hash join in gStore, which first builds a hash index on the common query variables of the smaller result set, then probes it by each result in the larger set. We can thus estimate the cost of a binary join as follows:

$$\begin{gathered} cost(\text{BinaryJoin}(V_1, V_2)) = \\ 2 \times \min(card(V_1), card(V_2)) + \max(card(V_1), card(V_2)) \end{gathered} \tag{3}$$

where the first operand estimates the cost of building the hash index, and the second estimates the cost of probing it.

3.5 Sampling-based Cardinality Estimator

A key ingredient in our cost model is the estimated size of intermediate results, i.e., cardinality, of a subquery. Given a subquery $Q(V)$ with query vertex set V, we denote its cardinality by $card(V)$. We aim for the estimate to be as accurate and stable as possible.

Our cardinality estimator mainly uses sampling methods, which are quite stable and usually more accurate than statistics methods like histograms [3]. Concretely, it consists of the following two steps:

(1) **Sampling One Variable.** For every query vertex in the query $Q(V)$, we draw a certain number of samples from the candidate result list (the size of which is denoted as `cand_size`) after constant filters. The number of samples must be carefully chosen: too many samples incur a high overhead, while too few samples cannot guarantee accuracy. So we choose the number of samples $|S|$ by the formula below:

$$|S| = \begin{cases} \text{cand\_size} & \text{cand\_size} < 50 \\ 50 & 50 \leq \text{cand\_size} < 100 \\ 11 \times \lfloor \ln(\text{cand\_size}) \rfloor & \text{cand\_size} \geq 100 \end{cases} \tag{4}$$

This gives a continuous function, which preserves the stability and smoothness of sampling when the candidate result size varies.

(2) **Extending on Samples**. We can estimate the cardinality $card(V_k)$ of subquery $Q(V_k)$ by extending on samples of subquery $Q(V_{k-1})$'s result. Algorithm 2 describes such a procedure. Because of the bottom-up structure of dynamic programming, at the time of estimating $card(V_k)$, $card(V_{k-1})$ and its samples $S(V_{k-1})$ have already been obtained. Therefore, we can directly extend the sampled results by checking whether there is a v_k meeting the connection conditions of the query and count the sample result size as $count$. Then we can calculate $card(V_k)$ by the equation in Line 7 of Algorithm 2.

To speed up the estimation, we also use the sampling technique (Line 6). However, different from handling a single variable, here we have to perform stream sampling, i.e., we do not yet know the size of the expanded result set during the sampling process. We use a method combining reservoir sampling and probabilistic sampling to reduce fluctuation. Note that although multiple plans may be enumerated for a V_k in Algorithm 1, we only need to estimate its

Algorithm 2: Sampling-Based Cardinality Estimation

```
Input: Query $Q(V_Q)$, estimated cardinality $card(V_{k-1})$ of $V_{k-1}$,
       sampled result $S(V_{k-1})$ of $V_{k-1}$, next joined variable $v_k$
Output: Estimated cardinality $card(V_k)$, where $V_k = V_{k-1} \cup \{v_k\}$
1 count ← 0 ;
2 foreach sampled result $t_{k-1}$ of $S(V_{k-1})$ do
3     if $t_{k-1}$ can join node $v_k$ according to query $Q(V_Q)$ then
4         foreach result $t_k$ of $extend(t_{k-1}, v_k)$ do
5             count ← count + 1;
6             choose whether to save $t_k$ to the samples of $V_k$ ;
7 $card(V_k) \leftarrow \max(\text{count} \times card(V_{k-1})/|S(V_{k-1})|, 1)$;
8 return $card(V_k)$
```

cardinality once we generate a plan for it, eliminating redundant computation.

3.6 DFS Execution for Limit-k Queries

The above descriptions are for normal query optimization. However, for limit-k queries, the size of demanded results (the `LIMIT` value) is usually much smaller than that of the full results. So we need to adjust our optimization strategies for limit-k queries.

Execution mode. The executor adopts the *BFS* mode for normal queries, starting with the full candidate list and traversing the plan tree in a breadth-first fashion. When handling limit-k queries, however, it switches to the *DFS* mode, which has the following characteristics suitable for limit-k:

- Fine granularity. Instead of starting with the full intermediate result set, DFS traverses the plan tree once with each candidate to save much computation to get the desired k results.
- Depth-first order. Depth-first traversals of the plan tree return valid results each time the root node is hit, rather than only at termination in BFS.

Plan enumerator. In our plan enumerator, we turn off the binary join optimization for DFS execution (Line 8 in Algorithm 1) because binary join needs its two children's full results as input which are unavailable in DFS.

Cost model. The last adjustment is our cost model. Since our target is to obtain k results, if a path in the plan tree cannot generate any results, we wish to discover this mismatch as early as possible. Also, the more outputs upper joins produce, the better. So we model the cost of a DFS plan with the formula below:

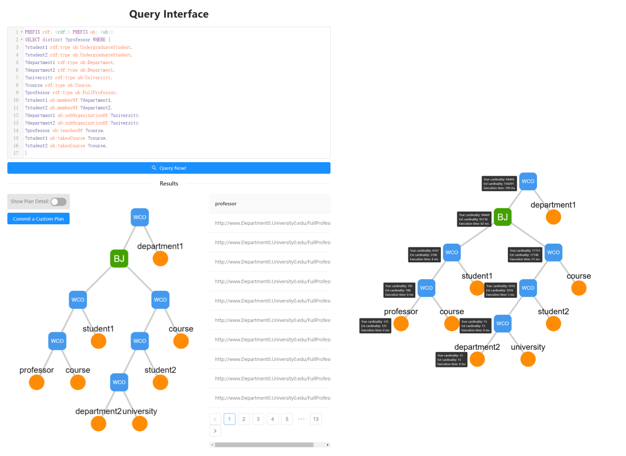
$$cost_{DFS}(T(Q)) = \sum_{O \in T(Q)} depth(O) \times cost_{DFS}(O) \tag{5}$$

where the $cost_{DFS}(O) = card(V_k)/card(V_{k-1})$ for a WCO join from V_{k-1} to V_k.

4 DEMONSTRATION

In this section, we demonstrate the functions of our optimizer. The first is plan tree visualization, and the second is interactive query plan generation and execution.

(1) **Plan Tree Visualization.** Users can input the query through the interface and click the "Query" button to see the query results, with the execution plan tree in the form mentioned in section



```
PREFIX rdf: <rdf:> PREFIX ub: <ub:>
SELECT distinct ?professor WHERE {
?student1 rdf:type ub:UndergraduateStudent.
?student2 rdf:type ub:UndergraduateStudent.
?department1 rdf:type ub:Department.
?department2 rdf:type ub:Department.
?university rdf:type ub:University.
?course rdf:type ub:Course.
?professor rdf:type ub:FullProfessor.
?student1 ub:memberOf ?department1.
?student2 ub:memberOf ?department2.
?department1 ub:subOrganizationOf ?university.
?department2 ub:subOrganizationOf ?university.
?professor ub:teacherOf ?course.
?student1 ub:takesCourse ?course.
?student2 ub:takesCourse ?course.
}
```

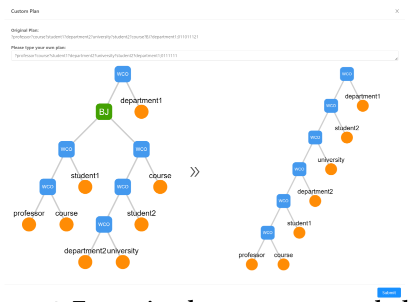
```
professor
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
http://www.Department0.University0.edu/FullProfes
< 1 2 3 4 5 ··· 13 >
```

(a) Snapshot of interface (b) Plan detail presentation

Figure 3: Plan tree visualization

3.2. Figure 3(a) shows a snapshot of our interface. We represent variables, WCO joins and binary joins as orange circles, blue squares with "WCO" labels and green squares with "BJ" labels, respectively. Furthermore, for normal queries, we can click "Show Plan Detail" to see the optimization information such as execution time, the estimated cardinality and true cardinality of each join operation, shown in Figure 3(b).

(2) **Interactive query plan generation and execution.** Users can generate any execution plans based on the operations we provide. The validity of the plan is checked by our system. If valid, our system will materialize and execute it. For example, Figure 4 shows a query plan provided by our optimizer on the left and an alternative pure-WCO join plan generated by a user on the right.



Original Plan:
?professor?course?student1?department2?university?student2?course?BJ?department1;011011121

Please type your own plan:
?professor?course?student1?department2?university?student2?department1;0111111

Figure 4: Execution by user-generated plan

5 CONCLUSION

In this demonstration, we propose a new cost-based optimizer called gCBO for graph databases, which adopts a dynamic programming-based plan enumerator and a sampling-based cardinality estimation strategy to tackle the challenges in query optimization. We also demonstrate an interactive component for gCBO that enables users to investigate and participate in the query execution process. We have implemented gCBO in the open-source graph database gStore.

ACKNOWLEDGMENTS

This work was supported by NSFC under grant 61932001 and U20A20174. The corresponding author of this paper is Lei Zou (zoulei@pku.edu.cn).

REFERENCES

[1] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767.

[2] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 query language. *W3C recommendation* 21, 10 (2013), 778.

[3] Kyoungmin Kim, Hyeonji Kim, George Fletcher, and Wook-Shin Han. 2021. Combining Sampling and Synopses with Worst-Case Optimal Runtime and Quality Guarantees for Graph Pattern Cardinality Estimation. In *Proceedings of the 2021 International Conference on Management of Data*. 964–976.

[4] 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.

[5] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704.

[6] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. 37–48.

[7] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record* 42, 4, 5–16.

[8] Li Zeng and Lei Zou. 2018. Redesign of the gStore system. *Frontiers of Computer science* 12, 4 (2018), 623–641.