# Converged Optimizer for Efficient Join Order Optimization

## ABSTRACT

## 1 INTRODUCTION

SQL/Property Graph Queries is Part 16 of SQL 2023 Graphs are presented as views, and the vertices and edges in the graphs are represented as tables For queries, the graph queries and relational queries can be expressed in one statement.

Example 1. *An example about how to write the sql/pgq query.*

Query optimization is crucial for query processing. The optimizer can significantly influence the efficiency of query processing. There are already many works about relational query optimizers, and some works about graph query optimizers. However, neither of them are suitable for SQL/PGQ queries. Because they can only optimize the queries from the relational perspective or the graph perspective, but not both. In this paper, we are going to propose a new converged framework for query optimization of SQL/PGQ statements.

There are mainly three challenges.

**Challenge 1. Relational optimizers cannot be used to optimize graph queries directly (or with limited efficiency)**. It is true that the vertices and edges in graphs can be represented as corresponding tables in relational databases, and the paths in the graphs can be translated into join operators in relational algebra. However, since the relational optimizers only take the relational operators into consideration, some efficient graph operators (e.g., get edge, get vertex, get neighbors with graph indices) are ignored in the process of optimization. Therefore, the search space is artificially reduced, the best physical plan are likely to be missed. An intuitive idea is to replace some relational operators with graph operators after the physical plans are obtained (e.g., replace some join operators with getV/getE/getNeighbor in graph operators) to take advantage of the benefits brought about by graph operators. Then, the obtained new plan are probabily not the optimizal plan, and the estimated costs are inaccurate.

Example 2. *An example about replace join with getV/getE/getNeighbor, or the example of duckdb, whether to indicate more constraints (due to the unawareness of getNeighbor)*

**Challenge 2. Graph optimizers sometimes cannot be used to optimize relational queries directly**. Relational queries cannot always be expressed as graph queries with no effort. When the tables in a relational query do not possess the semantics of vertices and edges, graph optimizers cannot be applied to optimize such queries. An example is as follows:

Example 3. *An example about: relation query for [p1]-join-[p2]-join-[p3] where, person p1, p2, and p3 have the same birthday Then, none of the tables can represent the edges, and the graph optimizer cannot be applied. An alternative is to create a new table* ***HAS_SAME_BIRTHDAY*** *= (id, person1id, person2id), each of whose record represents two persons having the same birthday. Then, the query should be converted to [p1]-join-[hsb]-join-[p2]-join=[hsb]-join-[p3]. However, it is not practical to create new tables in the process of query optimization.*

Moreover, relational query optimization has undergone numerous years of research and has accumulated a significant body of research findings. Therefore, it would be unwise to abandon relational query optimization in favor of direct graph query optimization.

**Challenge 3. How to ensure worst-case-optimal joins (WCOJs)**. Graph queries can be much more complex than relational queries, and multiple joins are common in graph queries. Then, it is necessary to support WCOJs in the optimizer, which can reduce the complexity of

In this paper, we propose a new converged optimization framework for SQL/PGQ. It can optimize SQL/PGQ statements for query. Specifically, the framework first optimize the graph queries in the statement and generate the corresponding graph physical plans. [Since there may be join operators in the graph physical plans, the plan can be considered to be connected with join operators. Then, the plan is combined with the left relational queries, the new query is optimized again with the relational optimizer.]

The contributions of this paper is mainly as follows:

(1) To the best of our knowledge, this is the first optimization framework for SQL/PGQ. Property graphs are represented as views in SQL/PGQ, and vertices and edges are associated with tables in the relational databases. Then, it is crucial to offer the converged query optimizer efficient for both relational and graph queries.

(2) The framework is the first to [unify] the inputs and outputs of the graph optimizer and relational optimizer based on the graph relational algebra, and propose the nested optimization strategy (abbr. NOS) for SQL/PGQ queries. In detail, given a SQL/PGQ query, NOS first optimizes the graph queries with the graph optimizer. The output plan is then optimized together with the relational queries by the relational optimizer.

(3) Theoretical analysis on the complexity of the optimization framework is conducted. The obtained theorems prove that for graph queries, the join order optimization with a graph optimizer can be exponentially faster than that with a relational optimizer. It theoretically confirms that relational optimizer is usually not suitable for graph queries, and it is indispensable for the existence of a converged optimization framework.

(4) Extensive experiments are conducted to show the efficiency of the proposed converged query optimization framework. The experimental results show that the framework can be ?× faster than the baselines.

The rest of this paper is organized as follows.

## 2 PRELIMINARIES

### 2.1 Data Model

Let $L$ be a finite set of labels, $D = \bigcup_i D_i$ be the union of atomic domains $D_i$, and $\epsilon$ be the NULL value. We define a *property graph* as $G = (V, E, \lambda, \mathcal{L}, \mathcal{T}, P_v, P_e)$, where

- $V$ is a finite set of vertices,

- $E$ is a finite set of edges,
- $\lambda : E \mapsto V \times V$ connects each edge $e$ with a tuple $(s, t)$ of source and target vertices,
- $\mathcal{L} : V \mapsto 2^L$ assigns a set of labels to each vertex,
- $\mathcal{T} : E \mapsto L$ assigns a label to each edge,
- $P_v$ is a set of vertex properties, where $p_v^i : V \mapsto D_i \cup \{\epsilon\}$ is a partial function that assigns a property value in $D_i$ to each vertex. In particular, if a vertex $v$ does not have the property $p_v^i$, $p_v^i(v) = \epsilon$,
- $P_e$ is a set of edge properties, where $p_e^j : E \mapsto D_j \cup \{\epsilon\}$ is a partial function that assigns a property value in $D_j$ to each edge. In particular, if an edge $e$ does not have the property $p_e^j$, $p_e^j(e) = \epsilon$.

Let $U = \{a_1, a_2, \ldots, a_n\}$ be a finite set of attributes, then $S = (a_1, a_2, \ldots, a_n)$ is called a schema over $U$. The attributes of $S$ is denoted as $\text{attr}(R) = U$. The value of each attribute $a \in \text{attr}(S)$ comes from specific domain, denoted as $\text{Dom}(a)$.

Given a property graph $G$, a graph schema is such a schema $S$ that $\forall a \in \text{attr}(S)$, $\text{Dom}(a) \subseteq V \cup E \cup D$. In other words, each attribute of a graph schema is either a vertex, or an edge, or data from arbitrary domain. A relation $R$ over a graph schema $S$ (i.e. $\text{sch}(R) = S$) is called a graph relation. For simplicity, we denote $\text{attr}(R)$ in short for $\text{attr}(S)$ with $\text{sch}(R) = S$ to retrieve the attributes of a graph relation $R$. We write $R.a$ to access a given attribute $a$ in the relation $R$.

Given a graph relation $R$, if $a \in \text{attr}(R) \subseteq V \cup E$, we can further access the property $p$ on the vertex/edge attribute via $p(R.a)$ (or $p(a)$ if the relation $R$ is clear in the context). Particularly, we use Id and Label to denote the built-in properties of the globally unique identifier and label of a vertex/edge. To clarify ambiguity, the term "attribute" always refers to the attribute of a relation, while the term "property" always refers to the property of a graph element in this article.

There is a unique concept of *path* while traversing a graph data structure. A path is an ordered sequence of edge segments that have been traversed in the graph, where each edge segment $\varepsilon = (sv, e, tv)$, is a 3-tuple with $e$ as the edge field, and $sv$ and $tv$ as the source and target vertices of the edge. The number of edge segments that a path $p$ contains is the length of the path, denoted as $|p|$. For example, $p = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k\}$ $(\varepsilon_i = (sv_i, e_i, tv_i))$ refers to a path that go through the edges of $e_1, \ldots, e_k$, whose length is $k$. Although path is typically represented in a simpler form as either the vertex sequence (only the vertices it passed) or the edge sequence (only the edges it passed), we maintain the edge segment in this article to allow flexible manipulations on the path, e.g. path unwinding.

## 2.2 Graph Relational Algebra

Some graph-related operators are also defined in the graph relational algebra. Typical operators are shown as follows.

A list of graph-related operators.

# 3 CONVERGED GRAPH RELATIONAL OPTIMIZER

## 3.1 Overview of the Framework

A system overview figure and some introduction

## 3.2 Leveraging Benefits of Both Relational and Graph Optimizers

## 3.3 Detailed Optimizations

CBO strategies, RBO strategies

ExtendedIntersection

Wort-Case Optimal

# 4 SUPERIORITY OF THE GRAPH OPTIMIZER ON GRAPH QUERIES

In this section, we prove that graph optimizers have better performance than relational optimizers. Besides, since graph optimizers cannot always be used to optimize relational queries, it confirms the necessity of proposing the converged query optimization.

As one of the most widely adopted optimizers for relational databases, we use Calcite [1, 3] as a representative of relational JOPTs and analyze its efficiency of join order optimization. For the converged JOPT, we use GLogue [2] as the graph JOPT, and integrate it with Calcite.

A more simple condition is first considered. That is, all the tables in the query can represent vertices or edges. Then, it is supposed that there are $n+m$ tables are joined together, where $n$ tables represent vertices while $m$ tables represent edges. Suppose that there are $t$ implementation methods for join. The complexities of optimizing the join order with Calcite and the converged JOPT are analyzed respectively as follows. For simplicity, the time complexities are evaluated by the number of physical plans generated by the JOPTs.

Theorem 1. *The time complexity of join order optimization with Calcite is at least* $O(\frac{4^{m+n-1}}{m+n} t^{m+n-1})$.

Proof. The $n+m$ tables joined together can form a graph, where the tables represent vertices in the graph, and if there is a join condition specifying the equivalence of two columns in two two tables, there is an edge between these two tables. Then, the complexity of join order optimization with Calcite can be estimated as the number of possible physical plans that can be generated.

To avoid cross product, except for the first table, the selected tables to be be joined with the already chosen tables should be neighbors of the chosen tables in the graph. Therefore, the join order can be represented as a spanning tree in the graph. By computing the number of physical plans can be generated according to each spanning tree, the total number of physical plans can be obtained. However, same phyical plans may be obtained according to different spanning trees. For example, suppose a graph is a rectangle with four vertices and four edges. It may be constructed for a query like:

SELECT ... FROM $A, B, C, D$ WHERE $A.1 = B.1$
AND $B.2 = C.1$ AND $C.2 = D.1$ AND $D.2 = A.2$.

Then, the spanning tree with edges {AB, BC, CD} and the one with edges {AB, BC, AD} can generate same physical plans. Consequently, the summation of the number of physical plans corresponding to all the spanning trees is larger than the actual number of physical plans. In this proof, we compute the number of physical plans for one spanning tree, and this number is the lower bound of the number of physical plans that can be generated.

Given a spanning tree with $k$ edges (i.e., $k+1$ vertices representing tables) and only one leaf node, the number of logical plans corresponding to the spanning tree is roughly

$$\begin{aligned} c(k) &= 2 * (c(0)c(k-1) + c(1)c(k-2) + \cdots + c(k-1)c(0)) \\ &= 2\Sigma_{i=0}^{i=k-1} c(i)c(k-1-i), \\ &\text{where } c(0) = 1. \end{aligned}$$

With the generating function, it is obtained that

$$c(k) = \frac{2^k}{k+1} C(2k, k).$$

Since $k$ is the number of edges, and $k = m + n - 1$ in the spanning tree. Thus, the number of logical plans w.r.t. a spanning tree is

$$\frac{2^{m+n-1}}{m+n} C(2m+2n-2, m+n-1) \geq \frac{4^{m+n-1}}{m+n}.$$

Then, the number of physical plans is at least $\frac{4^{m+n-1}}{m+n} t^{m+n-1}$, so is the complexity of join order optimization with Calcite.

In conclusion, Theorem 1 is correct. □

Lemma 2. *Given $n$ tables, the time complexity of join order optimization with Calcite has an upper bound of $O(\frac{(2n-2)!}{(n-1)!} t^{n-1})$.*

Proof. The upper bound of the time complexity of join order optimization is achieved when there is a condition between any two of the $n$ tables. Because at that time, the tables can be joined in any order.

Since each join order corresponds to a full binary tree with $2n-1$ nodes, the problem is to count the number of possible full binary trees. Similar to Catalan number, the number of full binary trees is $O(C(2n-2, n-1) - C(2n-2, n))$. For each full binary tree, there are $n!$ ways to set the leaf nodes. Then, the number of generated physical plans is $O(\frac{(2n-2)!}{(n-1)!} t^{n-1})$. □

Theorem 3. *The time complexity of join order optimization with the converged JOPT is smaller than $3^n$ if all the tables participant in join can represent vertices or edges.*

Proof. Since the $n+m$ tables represent vertices and edges respectively and the join among them can be represented as a graph query, the converged JOPT optimizes the join order with the graph optimizer, i.e., GLogue.

As GLogue ensures worst-case optimality and the considered patterns are all induced subgraphs, the time complexity of join order optimization with GLogue is not related to the number of edges (i.e., $m$). Because JOOP is reduced to a variant of shortest path problem, the time complexity of join order optimization is $O(\mathcal{E})$, where $\mathcal{E}$ is the number of edges in GLogue. In detail,

$$\begin{aligned} O(\mathcal{E}) &= C(n, n-1) * (2^{n-1} - 1) + C(n, n-2) * (2^{n-2} - 1) \\ &\quad + \cdots + C(n, 1) * (2^1 - 1) \\ &= 3^n - 2^{n+1} + 1 \\ &< 3^n. \end{aligned}$$

In conclusion, Theorem 3 is correct.

□

Based on the complexity analysis in Theorem 1 and Theorem 3, it is found that when the tables represent vertices and edges,

$$\begin{aligned} \frac{\text{Time Complexity of Calcite}}{\text{Time Complexity of the Converged JOPT}} &= \frac{\frac{4^{m+n-1}}{m+n} t^{m+n-1}}{3^n} \\ &\geq 2^{m-3} (\frac{4}{3})^n t^{m+n-1}. \end{aligned}$$

Therefore, it suggests that the converged JOPT is exponentially faster than Calcite for graph-like join order optimization. The results also indicates that integrating graph optimizers into relational optimizers can improve the efficiency of join order optimization significantly.

Then, if there are some tables that cannot represent vertices or edges, the order of such tables cannot be optimized with graph optimizers, and the comparion becomes more complicate. For example, suppose five tables form a clique, and then these tables cannot be vertices or edges in a graph. Specifically, according to the dataflow shown in Fig. **??**, the join order of these left tables are optimzed by relational optimizers such as Calcite. Let the number of left tables be $s$, and together with the table obtained by the join of the tables optimized with graph optimizer, Calcite needs to optimize the join order among $s+1$ tables.

When $s = 0$, all the tables represent vertices or edges, and the time complexity of the converged JOPT is exponentially smaller than that of Calcite as analyzed as above.

When $s = 1$, only one table (saying $T_1$) does not represent a vertex or an edge, and Calcite optimizes the join between two tables. One of these two tables is $T_1$, and the other is the table obtained by the join of the tables optimized with graph optimizer. Then, we have

$$\begin{aligned} &\frac{\text{Time Complexity of Calcite}}{\text{Time Complexity of the Converged JOPT}} > \\ &\qquad \frac{\frac{2^{m+n-1}}{m+n} C(2m+2n-2, m+n-1) t^{m+n-1}}{3^{\hat{n}} + \frac{(2s)!}{(s)!} t^s} \\ &\qquad = \frac{\frac{2^{m+n-1}}{m+n} C(2m+2n-2, m+n-1) t^{m+n-1}}{3^{\hat{n}} + 2} >> 1, \end{aligned}$$

where $\hat{n}$ is the number of tables representing vertices. It indicates the superiority of the converged JOPT.

When $s = m+n-1$ or $p = m+n$, at most one table can represent vertices or edges, and the converged JOPT degenerates to Calcite, and has the same efficiency as it. A typical example is when there is a condition between any two of these $m+n$ tables.

The results show that the converged JOPT is always superior to relational JOPTs. To be more specific, we prove the superiority of the converged JOPT more theoretically.

Lemma 4. *Let $\mathcal{JN}_c(V_s)$ represent the number of possible physical plans of joining tables in table set $V_s$ with Calcite. Suppose $n$ tables (denoted as table set $V$) are joined, and $V = (V_1 - u) \cup V_2, V_1 \cap V_2 = \emptyset$. Specifically, $u \in V_1$ is a table representing the results of joining the tables in $V_2$. For each table $t_2 \in V_2$, if there is a join condition between $t_2$ and a table $v$ in $V_1$, the same join condition exists between $u$ and $v$. Then, we have $\mathcal{JN}_c(V) \geq \mathcal{JN}_c(V_1) * \mathcal{JN}_c(V_2)$.*

Proof. For a physical plan generated by joining tables in $V_1$ (denoted as $p_1$) and a plan generated by joining tables in $V_2$ (denoted as $p_2$), by replacing $u$ in $p_1$ with $p_2$, we can generate a physical plan of joining tables in in $V$. Besides, since the tables in $p_1$ cannot be interchanged with tables in $p_2$, more physical plans can be generated by joining tables in $V$. In conclusion, we have $\mathrm{JN}_c(V) \geq \mathrm{JN}_c(V_1) * \mathrm{JN}_c(V_2)$. □

Theorem 5. *The time complexity of join order optimization with the converged JOPT is always smaller than that with Calcite.*

Proof. Denote the set of tables that cannot represent vertices and edges by $S$. Besides, denote the set of tables that represent vertices and edges by $R$, and denote the table obtained by joining the tables in $R$ by $T_r$. Then, we have $S_r = S \cup T_r$. Moreover, let $s = |S|$ and $r = |R|$ represent the size of tables set $S$ and $R$, respectively, and let $s_v$ be the number of tables in $S$ representing vertices. Denote the number of possible physical plans of joining tables in $R$ with GLogue by $\mathrm{JN}_g(R)$.

Specifically, according to Theorem 1 and Theorem 3, we have $\mathrm{JN}_g(R) < 3^{s_v} \leq \frac{4^{r-1}}{r} t^{r-1} \leq \mathrm{JN}_c(R)$. Note that $T_r$ corresponds to $u$ in Lemma 4. Based on Lemma 4, we have $\mathrm{JN}_c(S \cup R) \geq \mathrm{JN}_c(S_r) * \mathrm{JN}_c(R) \geq \mathrm{JN}_c(S_r) * \mathrm{JN}_g(R)$. In conclusion, Theorem 5 is correct, and the converged JOPT always has smaller time complexity than Calcite. □

There are mainly three reasons contributing to the efficient performance of the converged JOPT: (1) Different implementations of the join operators are not considered in the converged JOPT, because the neighbors of vertices can be efficiently accessed with the graph indices. (2) In the converged JOPT, only the number of vertices influence the complexity of join order optimization, while the complexity is determined by both the numbers of vertices and edges in Calcite. (3) The converged JOPT can take isomorphism into consideration in optimization and further reduce the search space, while Calcite does not consider optimization related to isomorphism.

# 5 EVALUATION

# 6 RELATED WORK

# 7 CONCLUSIONS

# REFERENCES


[1] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[2] Longbin Lai, Yufan Yang, Zhibin Wang, Yuxuan Liu, Haotian Ma, Sijie Shen, Bingqing Lyu, Xiaoli Zhou, Wenyuan Yu, Zhengping Qian, Chen Tian, Sheng Zhong, Yeh-Ching Chung, and Jingren Zhou. 2023. GLogS: Interactive Graph Pattern Matching Query At Large Scale. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 53–69. https://www.usenix.org/conference/atc23/presentation/lai

[3] Yongwen Xu. 1998. EFFICIENCY IN THE COLUMBIA DATABASE QUERY OPTIMIZER.