Variable-length Path Query Evaluation based on Worst-Case Optimal Joins

Mingdao Li
Hunan University
Changsha, China
limingdao@hnu.edu.cn

Peng Peng
Hunan University
Changsha, China
hnu16pp@hnu.edu.cn

Zheyuan Hu
Hunan University
Changsha, China
huzheyuan@hnu.edu.cn

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

Zheng Qin
Hunan University
Changsha, China
zqin@hnu.edu.cn

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00256

***Abstract*—Variable-length path queries are essential for finding paths in a graph that adhere to a specified length constraint, utilizing only edges with labels from a restricted subset of the edge labels. These queries play a crucial role in graph analytics and are supported by practical graph query languages like Cypher in property graph systems and SPARQL 1.1 in RDF graph systems. In this paper, we present a novel solution for efficient evaluation of variable-length path queries, based on worst-case optimal joins. Our solution's core relies on a jumping-like worst-case optimal join technique, allowing us to select a query vertex order that differs completely from existing graph systems based on worst-case optimal joins. Furthermore, we introduce a cost-based dynamic programming optimizer that combines traditional and jumping-like worst-case optimal join techniques. We also propose an optimization technique to leverage intra-query parallelism during query evaluation. Through extensive experiments conducted on numerous synthetic and real RDF and property graphs, we demonstrate that the proposed technique achieves excellent performance.**

I. Introduction

As a general data structure, graphs have been widely used in many fields including Internet, chemistry, biological information, social networks and so on. Resource Description Framework (RDF) and property graphs are two well-known ways to model graphs, and a number of graph systems have been proposed to manage graphs in the two ways.

In this paper, we study a fundamental type of query in both RDF and property graph systems, the problem of answering variable-length path queries. This problem is to find pairs of vertices in a graph connected by a path whose labels (i.e., the labels of edges in the path) is in a subset of the user-specified edge labels and length meets a range constraint.

The problem can be used in many real applications. For example, in social networks, we may want to match possible friends of friends and friends of friends of friends and return them all in the same collection. Then, the query of finding all the people that can be reached from other people by a path between 2 and 3 occurrences of KNOWS can be represented as the following queries in two well-known graph query languages, Cypher and SPARQL.

Example 1.1: Neo4j [2] is a graph database system for managing the property graph, and Cypher [12] is a declarative graph query language to query the property graph proposed by Neo4j. In Cypher, variable-length path queries are supported. For example, the following query is to find all the people that can be reached from other people by a path between 2 and 3 occurrences of KNOWS.

$$Match\, path = (x)-[: KNOWS{*}2..3]->(y)\;\; return\, path;$$

□

Example 1.2: RDF is a model that is widely used for publishing data in the web, and SPARQL is the structural query language. In the newest version of SPARQL, SPARQL 1.1, W3C propose a new way named *property paths* to extend matching of triple pattern to arbitrary length paths. Many RDF systems, like Jena [1] and Virtuoso [4], can also support the variable-length path queries. For example, the following query is to find all the people that can be reached from other people by a path between 2 and 3 occurrences of foaf:knows.

$$SELECT\ ?x\ ?y\ WHERE\ \{?x\ foaf:knows\{2,3\}\ ?y\}$$

□

Formally, a *variable-length path query* can be represented $P^{[m,n]}(L_E^Q)$, where L_E^Q is the set of edge labels, and $[m,n]$ is the a range of path length constraint. Given a graph, a variable-length path query $P^{[m,n]}(L_E^Q)$ is to find all paths, of which the lengths are not larger than n and not smaller than m and the edge labels in L_E^Q.

In the context of variable-length path queries, existing graph database systems, whether they are RDF graph systems [1], [3], [4], [14], [30] or property graph systems [2], [16], treat path queries as general queries without specifically optimizing their characteristics. In the case of a variable-length path query, all query edges are homogeneous, allowing for the reuse of intermediate results obtained from fewer query edges, a feature that is often overlooked in current optimization approaches.

Furthermore, except Jena-LFJ [14] and Graphflow [16], most graph database systems do not incorporate the technique of worst-case optimal joins into their frameworks. Worst-case optimal join [23], [22] is a type of join technique that can theoretically guarantee the query result size within an *AGM bound* [8]. The most renowned worst-case optimal join algorithms [23], [21] typically adopt a strategy where they first establish a query vertex order and then evaluate the query vertex by vertex. However, these algorithms tend to assume that any prefix of the query vertex order is connected, overlooking potential orders where some prefixes of the query vertex order might not be connected when we consider the variable-length path queries.

In this paper, we introduce a novel approach for handling variable-length path queries that does not require the construction of any dedicated indices while still ensuring worst-case optimality. Specifically, for a given path query of length k, our approach can also guarantee that there are at most $|E|^{\lfloor \frac{k}{2} \rfloor + 1}$ results, where $|E|$ represents the number of edges in the graph.

First, we propose a new jumping worst-case optimal join technique. For a path query of n edges, our proposed join technique can reuse the intermediate results of some path queries of fewer edges and avoid generating the results of some other path queries of fewer edges while guaranteeing worst-case optimality. In other words, our method can pick a query vertex order, where some prefixes of the query vertex order is not connected. In contrast, because traditional worst-case optimal join techniques tend to require that any prefix of the query vertex order is connected, it needs to generate a sequence of results for path queries from 1 to $n-1$ edges to obtain the results of a path query of n edges.

Then, to mix the basic and jumping worst-case optimal join techniques, we design a cost-based optimizer. The cost model behind our optimizer is based on the classic database statistics, *selectivity factor*. We discuss how to estimate the selectivity factors of both basic and jumping worst-case optimal join techniques. Then, based on the cost model, a dynamic programming query plan generation algorithm is proposed to determine the optimal query plan. The query plan is represented as a directed acyclic graph (DAG). The DAG of a query plan indicates the potential of parallelism. We can find some parts in the DAG that are independent on each other, and then exploit the intra-query parallelism to further improve the performance of query evaluation.

To verify our proposed techniques, we embed them into both a property graph system, Graphflow [16] and an RDF graph system, gStore [30]. Then, we evaluate it across a large number of synthetic and real graphs.

In summary, our main contributions are as follows:

- We conduct an analysis of the AGM bound concerning variable-length path queries, and introduce a novel join technique known as the *jumping worst-case optimal join*. This technique can reuse intermediate results while ensuring the worst-case optimality.
- To ascertain the optimal plan that integrates basic and jumping worst-case optimal join techniques, we devise a cost-based dynamic programming query plan generation algorithm. The cost model is designed based on the structural characteristics of input graphs.
- The devised query plan is represented as a Directed Acyclic Graph (DAG), and we identify independent segments conducive to parallelism. Subsequently, we implement parallelism for the evaluation of these independent components within the query plan.
- Our proposed methodologies are integrated into existing property graph and RDF graph systems. Extensive experiments is done on both synthetic and real graphs to validate the effectiveness of our approaches.

II. Related Work

Recently, there have been a lot of graph systems proposed to store, process, and query graphs efficiently. They include RDF graph systems, like Jena [1], Virtuoso [4], RDF4J [3] and gStore [30], and property graph systems, like Neo4j [2], TigerGraph [10], GraphScope [11], Graphflow [16] and EmptyHeaded [5], [6]. Some of them support query languages like SPARQL, Cypher and GSQL [20] that contain the operator of variable-length path query, but few of them discuss how to design optimizations for variable-length path query.

Worst-case optimal join techniques [23], [21] have recently gained a lot of attention, which can guarantee the output size of a database query in the worst case. A few graph systems, like Jena-LFJ [14], EmptyHeaded [5], [6], Graphflow [16] and PatMat [17] also implement the worst-case optimal join. The main idea of the worst-case optimal join techniques in graph systems is to process subgraph matching by matching vertices in a query vertex order, and any prefixes of the query vertex order used in existing worst-case optimal join techniques are always connected. In this paper, we propose a jumping-like worst-case optimal join technique for variable-length path queries that can generate the query vertex order whose some prefixes are disconnected.

There is no studies focused on optimizing variable-length path queries. Existing studies primarily focus on label-constraint reachability queries [15], [28], [31], [27], [26], [25], [29], [9]. These studies are to determine the presence of a path from a source vertex to a target vertex, using only edges with labels within a predefined set. These studies do not consider the path length constraints and always build some extra indices for optimization. For example, Jin et al. [15] present a tree-based index, which consists of a spanning tree and a partial transitive closure of the graph; Zou et al. [28], [31] decompose the input graph into strongly connected components (SCC) and maintain the label sets in a SCC or among different SCCs; Valstar et al. [27] propose a landmark-based index that maintain the minimal label set connecting each landmark to other vertices; Peng et al. [26], [25] design a label-constrained 2-hop indexing techniques with some pruning rules and order strategies; Cai et al. [9] introduce two pruning techniques, degree-one reduction and an unreachable query filter, to further minimize the index size and expedite both index construction and query processing. Zeng et al. [29] address the problem in a distributed environment for processing large graphs.

III. Preliminaries

A. Problem Definitions

Definition 3.1: **(Graph)** A *graph* G is denoted as $\langle V(G), E(G), L_E, f^G \rangle$, where $V(G)$ is a set of vertices, $E(G) \subseteq V(G) \times V(G)$ is a set of directed edges, L_E is a set of edge labels and $f^G : E(G) \rightarrow L_E$ is a function that assigns each edge with one label. □

For an edge of $e = u \xrightarrow{l} v \in E$, it is considered to be directed from u to v, where u is called the head, v is called the tail and l is called the label of the edge. For a vertex v, the set of it incoming edges is called its set of incoming neighbor edges and denoted as $N^{-}(v)$; the set of it outgoing edges is called its set of outgoing neighbor edges and denoted as $N^{+}(v)$.

We define a sequence $\pi = v_0 e_1 v_1, ..., v_{k-1} e_k v_k$ to represent a path of length k between two vertices v_0 and v_k, where $\{v_0, v_1, ..., v_k\} \subseteq V$ and $\{e_1, e_2, ..., e_{k-1}\} \subseteq E$.

Given a set of labels $L_E^Q \subseteq L_E$, the set of edges with a property l ($l \in L_E$) is denoted as $E(L_E^Q)$.

In this paper, the graph can be stored in all kinds of graph-based data structures, including both adjacency list and adjacency matrix. The only need is to support to find out all matches of a query edge of a given label. Obviously, this need is a very basic operator and can be efficiently supported by all graph database systems.

Definition 3.2: **(Variable-length Path Query)** A *variable-length path query* is represented $P^{[m,n]}(L_E^Q)$, where L_E^Q is the set of edge labels, and $[m, n]$ is the a range of path length constraint.

Given a graph G, a variable-length path query $P^{[m,n]}(L_E^Q)$ is to find all paths, of which the lengths are not larger than n and not smaller than m and the edge labels in L_E^Q. In this paper, we stipulate that each returned path must be a simple path, wherein no two vertices are the same.

The set of result paths for $P^{[m,n]}(L_E^Q)$ is denoted as $[\![P^{[m,n]}(L_E^Q)]\!]$. □

A path query of a specific length constraint n is a special kind of variable-length path query, where $m = n$. We simplify it as $P^n(L_E^Q)$ and its results as $[\![P^n(L_E^Q)]\!]$.

Further, a path query without the length limit is denoted as $P^{[1,+\infty]}(L_E^Q)$ and its results as $[\![P^{[1,+\infty]}(L_E^Q)]\!]$. This is an extreme case of variable-length path query, which is widely used as a wildcard on the edge label constraint.

B. Worst-Case Optimal Joins

A recent study of Atserias, Grohe, and Marx [8] showed how to tightly bound the worst-case size of a join query using a notion called a *fractional edge cover*. The bound is often called the *AGM bound*.

To find the optimal join, the definition of hypergraph is also needed.

Definition 3.3: **(Hypergraph)** A *hypergraph* is a pair $H = (V(H), E(H))$, consisting of a nonempty set $V(H)$ of vertices, and a set $E(H)$ of subsets of $V(H)$, the hyperedges of H.□

Definition 3.4: **(Fractional Edge Cover)** A *fractional edge cover* of a hypergraph $H = (V(H), E(H))$ is a mapping $\psi : E \to [0, +\infty)$ such that $\sum_{e \in E, v \in e} \psi(e) \geq 1$ for all $v \in V$. The number $\rho = \sum_{e \in E} \psi(e)$ is the weight of ψ.

The *fractional edge cover number* $\rho^*(H)$ of H is the minimum of the weights of all fractional edge covers of H, where the fractional edge cover corresponding to $\rho^*(H)$ is denoted as ψ^*.□

In particular, there is a direct correspondence between a query and its hypergraph: there is a vertex for each attribute of the query and a hyperedge for each relation. We will go freely back and forth between the query and the hypergraph that represents it.

AGM [8] showed that if ψ is feasible, then it forms an upper bound of the query result size $|OUT|$ as follows:

$$|OUT| \leq \Pi_{e \in E(H)} |R_e|^{\psi^*(e)}$$

Generic Join [23], [21] is a worst-case optimal join algorithm that evaluates queries one vertex at a time. It first determines an query vertex order σ, and then matches the query one vertex by one vertex according to the order σ. In previous studies, they often assume the subquery induced by the first k query vertices in σ is connected. Then, the algorithm has the following two basic operators:

Scan: For the first two vertices in the order σ, they correspond to a single query edge, and are evaluated with a SCAN operator. The operator scans the data graph and find the edges matching the labels of these two vertices and its corresponding query edge. Last, the Scan operator outputs each matched edge as a 2-match.

Extension/Intersection (E/I): After the subquery induced by the first k ($k > 2$) vertices in the order σ have been evaluated, the E/I operator takes matches of k vertices as input and extends each match to one or more matches of $k+1$ vertices. Given a match μ of k vertices, let $\mu(i)$ be the vertex matching the i-th vertex $?x_i$ in σ ($i \leq k$). We extend μ by intersecting the forward adjacency list of $\mu(i)$ for each $?x_i \to ?x_{k+1}$ in the query and the backward adjacency list of $\mu(i)$ for each $?x_{k+1} \to ?x_i$ in the query. Let the result of this intersection be the extension set S of μ. The $(k+1)$-matches t0 produces are the Cartesian product of t0 with S.

C. Analysis of Variable-Length Path Query

Based on the AGM bound, we can prove that there are at most $|E(L_E^Q)|^{\lfloor \frac{n}{2} \rfloor + 1}$ solutions for a variable-length path query $P^n(L_E^Q)$, which will serve as inspiration for the development of the new jumping worst-case optimal join technique and subsequent complexity analysis in the following section. To prove this, we first prove the lemma as follows.

Lemma 3.1: Given a path $\pi = v_0 e_1 v_1, ..., v_{k-1} e_k v_k$ and its fractional edge cover ψ^* corresponding to $\rho^*(\pi)$, $\psi^*(e_1) = \psi^*(e_k) = 1$.

Proof: Here, we assume that the fractional edge cover corresponding to $\rho^*(\pi)$ is ψ^*. According the definition of fractional edge cover, for v_0, it only have one adjacent edge e_1, so $\psi^*(e_1) \geq 1$.

If $\psi^*(e_1) > 1$, then we can find out another fractional edge cover ψ' as follows.

$$\psi'(e_i) = \begin{cases} 1, if\ i = 1, \\ \psi^*(e_i), otherwise. \end{cases}$$

Obviously, $\sum_{i=1}^{k} \psi'(e_i)$ is smaller than $\sum_{i=1}^{k} \psi^*(e_i)$, which conflicts with the fact that $\rho^*(\pi) = \sum_{i=1}^{k} \psi^*(e_i)$ is the minimum of the weights of all fractional edge covers. Hence, $\psi^*(e_1) = 1$.

Similarly, we can prove that $\psi^*(e_k) = 1$. ■

Then, we can obtain the fractional edge cover number of a path with n edges as follows.

Theorem 1: Given a path $\pi = v_0 e_1 v_1, ..., v_{k-1} e_k v_k$, its fractional edge cover number $\rho^*(\pi)$ is $\lfloor \frac{k}{2} \rfloor + 1$.

Proof: We first prove that the lower bound of $\rho^*(\pi)$ is $\lfloor \frac{k}{2} \rfloor + 1$. Then, we prove that $\rho^*(\pi)$ can reach $\lfloor \frac{k}{2} \rfloor + 1$. Here, we assume that the fractional edge cover corresponding to $\rho^*(\pi)$ is ψ^*.

In the first case, k is an odd number. For each vertex v_i $(1 \leq i \leq k-1)$, according the definition of fractional edge cover, we have $\psi^*(e_i) + \psi^*(e_{i+1}) \geq 1$. Thus, we have

$$\sum_{i=1}^{k-1} (\psi^*(e_i) + \psi^*(e_{i+1})) \geq k - 1$$

$$\Rightarrow \psi^*(e_1) + \psi^*(e_k) + 2 \times \sum_{i=2}^{k-1} \psi^*(e_i) \geq k - 1$$

According to Lemma 3.1, $\psi^*(e_1) = \psi^*(e_n) = 1$. Thus,

$$\psi^*(e_1) + \psi^*(e_k) + 2 \times \sum_{i=2}^{k-1} \psi^*(e_i) + \psi^*(e_1) + \psi^*(e_k) \geq k - 1 + 2$$

$$\Rightarrow 2 \times \sum_{i=1}^{k} \psi^*(e_i) \geq k + 1$$

According to Definition 3.4, $\sum_{i=1}^{k} \psi^*(e_i) = \rho^*(\pi)$. Thus,

$$2 \times \rho^*(\pi) \geq k + 1 \Rightarrow \rho^*(\pi) \geq \frac{k+1}{2}$$

Note that, since n is an odd number, $\frac{k+1}{2} = \lfloor \frac{k}{2} \rfloor + 1$. Therefore, $\rho^*(\pi) \geq \lfloor \frac{k}{2} \rfloor + 1$.

Meanwhile, for an odd number k, to reach $\lfloor \frac{k}{2} \rfloor + 1$, we can define the mapping ψ^* as follows.

$$\psi^*(e_i) = \begin{cases} 1, if\ i\ is\ an\ odd\ number, \\ 0, otherwise. \end{cases}$$

In the second case, k is an even number. For each vertex $v_{2 \times i}$ $(1 \leq i \leq \frac{k-2}{2})$, we have $\psi^*(e_{2 \times i - 1}) + \psi^*(e_{2 \times i + 1}) \geq 1$. Thus,

$$\sum_{i=1}^{\frac{k-2}{2}} (\psi^*(e_{2 \times i - 1}) + \psi^*(e_{2 \times i + 1})) \geq \frac{k-2}{2}$$

$$\Rightarrow \sum_{i=2}^{k-1} \psi^*(e_i) \geq \frac{k-2}{2}$$

According to Lemma 3.1, $\psi^*(e_1) = \psi^*(e_k) = 1$. Thus,

$$\sum_{i=2}^{k-1} \psi^*(e_i) + \psi^*(e_1) + \psi^*(e_k) \geq \frac{k-2}{2} + 2$$

$$\Rightarrow \sum_{i=1}^{k} \psi^*(e_i) \geq \frac{k}{2} + 1$$

According to Definition 3.4, $\sum_{i=1}^{k} \psi^*(e_i) = \rho^*(\pi)$. Thus,

$$\rho^*(\pi) \geq \frac{k}{2} + 1$$

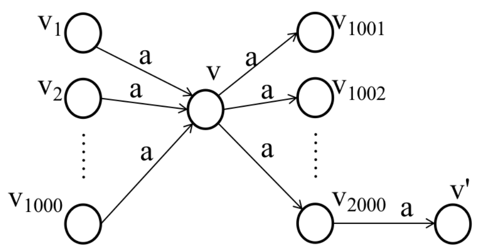


Fig. 1. An Extreme Example Graph

Note that, since k is an even number, $\frac{k}{2} + 1 = \lfloor \frac{k}{2} \rfloor + 1$. Therefore, $\rho^*(\pi) \geq \lfloor \frac{k}{2} \rfloor + 1$.

As well, for an even number k, we can reach $\lfloor \frac{k}{2} \rfloor + 1$ by defining the mapping ψ^* as follows.

$$\psi^*(e_i) = \begin{cases} 1, if\ i = 1\ or\ i = k, \\ \frac{1}{2}, otherwise. \end{cases}$$

■

Finally, we can prove the bound of a variable-length path query $P^n(L_E^Q)$ as follows.

Theorem 2: Given a graph $G = (V, E)$ and a variable-length path query $P^n(L_E^Q)$, there are at most $|E(L_E^Q)|^{\lfloor \frac{n}{2} \rfloor + 1}$ solutions.

Proof: As proved in Theorem 1, the fractional edge cover number of $P^n(L_E^Q)$ is $\lfloor \frac{n}{2} \rfloor + 1$, so the number of results for $P^n(L_E^Q)$ is at most $|E(L_E^Q)|^{\lfloor \frac{n}{2} \rfloor + 1}$ based on the conclusions in [8]. ■

IV. Jumping Worst-Case Optimal Joins

Let us first consider the path query $P^3(L_E^Q)$ of the length constraint 3. To better show its execution, we expand it in the following way.

$$?x_1 \xrightarrow{L_E^Q} ?x_2 \xrightarrow{L_E^Q} ?x_3 \xrightarrow{L_E^Q} ?x_4 \quad (1)$$

Existing worst-case optimal joins-based algorithms always extend a connected subquery by one query vertex adjacent to at least one vertex in the connected subquery. Thus, they always generate the results of $P^3(L_E^Q)$ by extending the results of $P^2(L_E^Q)$. However, according to Theorem 2, then the numbers of query results for both $P^3(L_E^Q)$ and $P^2(L_E^Q)$ are at most $|E|^2$. Then, we find that the number of results of $P^3(L_E^Q)$ and $P^2(L_E^Q)$ are asymptotically equivalent, but existing methods extend the results of $P^2(L_E^Q)$ to obtain the results of $P^3(L_E^Q)$. It indicates that there is much room for optimizations.

For example, let us consider an extreme graph as shown in Fig. 1. For path query $P^3(\{a\})$, we should first enumerate the results of $P^2(L_E^Q)$, the number of which is more than one million as illustrated in Fig. 2. However, most results of $P^2(L_E^Q)$ are unnecessary and cannot contribute to final results of $P^3(L_E^Q)$. There are only one thousand results for $P^3(L_E^Q)$.

However, we can evaluate $P^3(L_E^Q)$ in another way. We can first do two scans for both query edges $?x_1 \xrightarrow{L_E^Q} ?x_2$ and $?x_3 \xrightarrow{L_E^Q} ?x_4$. Then, we create a hash table of all of the matches of $?x_3 \xrightarrow{L_E^Q} ?x_4$ on the vertices matching query vertex $?x_3$.

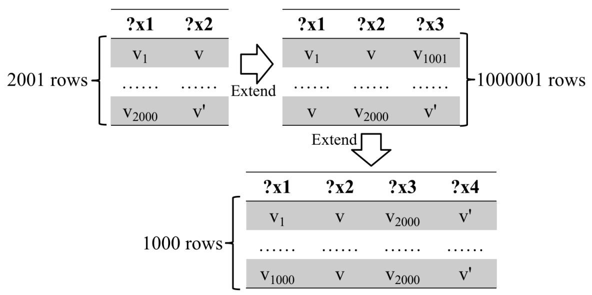


Fig. 2. Execution of Existing Worst-Case Optimal Join Technique

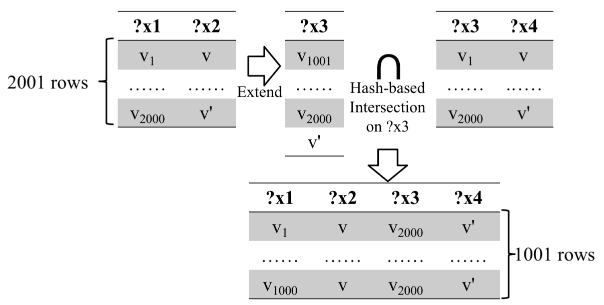


Fig. 3. Execution of Our Jumping Worst-Case Optimal Join Technique

When we do extension from the matches of $?x_1 \xrightarrow{L_E^Q} ?x_2$ to the vertices matching $?x_3$, if the vertices matching $?x_3$ are not contained in the hash table for $?x_3 \xrightarrow{L_E^Q} ?x_4$, we can skip the matches. This join technique is called the *jumping worst-case optimal join technique* in this paper.

Note that, the above worst-case optimal join technique is still a worst-case optimal join, because it still matches $?x_3$ at a time using the intersection between all edges adjacent to $?x_3$. Indeed, it is equivalent to evaluate the query using the query plan $\sigma = <?x_1, ?x_2, ?x_4, ?x_3>$. The technique can make the query plan $\sigma = <?x_1, ?x_2, ?x_4, ?x_3>$ be more efficient than existing worst-case optimal joins-based algorithms for the variable-length path queries, especially for the data graphs that contains many paths of two edges but few paths of three edges.

For the path query in Equation 1, existing worst-case optimal joins-based algorithms rarely generate a plan as $\sigma = < ?x_1, ?x_2, ?x_4, ?x_3 >$. This is also understandable, because for the query plan $\sigma = <?x_1, ?x_2, ?x_4, ?x_3>$ the previous algorithms should extend from the candidate matches of $?x_2$ and $?x_4$ and do intersection over the adjacent lists of candidate matches of $?x_2$ and $?x_4$. The candidate matches of $?x_4$ in this plan can be any vertices in the graph.

Fig. 3 shows the execution of the jumping worst-case optimal join technique for $P^3(\{a\})$ over the data graph in Fig. 1. Here, we first scan the data graph to get the matches of $?x_1 \xrightarrow{\{a\}} ?x_2$ and $?x_3 \xrightarrow{\{a\}} ?x_4$. These two query edge are the same, so they have the same set of matches and there are about two thousand matches. Then, we build up a hash table for the matches of $?x_3 \xrightarrow{\{a\}} ?x_4$, where the keys are the vertices matching $?x3$ and the values are the matches of $?x_3 \xrightarrow{\{a\}} ?x_4$. The hash table does not contain v_{1001}, v_{1002}, ..., v_{1998} or v_{1999}, so we can avoid the intermediate matches of two edges ending with them.

In addition, the characteristic of the variable-length path query can make the above technique be more useful. The definition of the variable-length path query require that the label of all edges in the matched paths should meet the same label constraint, which ensure that the matches of both query edges $?x_1 \xrightarrow{L_E^Q} ?x_2$ and $?x_3 \xrightarrow{L_E^Q} ?x_4$ are the same. Then, the generation of matches of both query edges $?x_1 \xrightarrow{L_E^Q} ?x_2$ and $?x_3 \xrightarrow{L_E^Q} ?x_4$ can finish by scanning the graph at one time.

Finally, we give the formal description of our join technique, named *jumping worst-case optimal join (J-WCO)* (Algorithm 1), which extends the basic extension/intersection operator in generic join for variable-length path queries. Given a path query $P^n(L_E^Q)$ of n edges, it can be divided into two subqueries, subquery $P^k(L_E^Q)$ of first k edges and subquery $P^{n-k-1}(L_E^Q)$ of last $n-k-1$ edges ($1 \leq k \leq n-1$). We first evaluate $P^k(L_E^Q)$ and $P^{n-k-1}(L_E^Q)$, and then build up a hash table of results of $P^{n-k-1}(L_E^Q)$ (Lines 1-4 in Algorithm 1). Then, we extend from the adjacent list of the matches of last variable in $P^k(L_E^Q)$ to get matches of the first variable in $P^{n-k-1}(L_E^Q)$ and check whether it is in the hash table (Lines 5-9 in Algorithm 1). The join technique skips one interval edge to join the results of $P^k(L_E^Q)$ and $P^{n-k-1}(L_E^Q)$ edges to form the results of a path query of n edges.

Algorithm 1: Jumping Worst-Case Optimal Join Algorithm

```
Input: Resuts of Two Subqueries $P^k(L_E^Q)$ and
       $P^{n-k-1}(L_E^Q)$ of $P^n(L_E^Q)$, $[\![P^k(L_E^Q)]\!]$ and
       $[\![P^{n-k-1}(L_E^Q)]\!]$, and we assume that $[\![P^{n-k-1}(L_E^Q)]\!]$
       is smaller than $[\![P^{n-k-1}(L_E^Q)]\!]$
Output: Results of $P^n(L_E^Q)$, $[\![P^n(L_E^Q)]\!]$
1  Recursively compute $[\![P^k(L_E^Q)]\!]$ and $[\![P^{n-k-1}(L_E^Q)]\!]$;
2  Initialize a hash map $ResMap$ for $[\![P^{n-k-1}(L_E^Q)]\!]$;
3  for each match $r$ in $[\![P^{n-k-1}(L_E^Q)]\!]$ do
4      Insert a mapping in $ResMap$, where the key is the
       match of the first variable $r[1]$ and the value is $r$;
5  for each match $r$ in $[\![P^k(L_E^Q)]\!]$ do
6      Extends $r$ to matches of $P^{k+1}(L_E^Q)$, denoted as
       $[\![P^{k+1}(L_E^Q)]\!]_r$;
7      for each match $r'$ in $[\![P^{k+1}(L_E^Q)]\!]_r$ do
8          if match of the last variable $r'[k+1]$ in $r'$ is in
           $ResMap$ then
9              Join $r'$ with $ResMap(r'[k+1])$ and add
               matches into $[\![P^n(L_E^Q)]\!]$;
10 Return $[\![P^n(L_E^Q)]\!]$;
```

Complexity Analysis. For space complexity, the jumping join technique requires constructing a hash table for either the results of $P^k(L_E^Q)$ ($[\![P^k(L_E^Q)]\!]$) or the results of $P^{n-k-1}(L_E^Q)$ ($[\![P^{n-k-1}(L_E^Q)]\!]$). In implementing the jumping join technique, we typically choose the smaller of $[\![P^k(L_E^Q)]\!]$ and $[\![P^{n-k-1}(L_E^Q)]\!]$ to construct the hash table, resulting in a space cost of $\min\{|[\![P^k(L_E^Q)]\!]|, |[\![P^{n-k-1}(L_E^Q)]\!]|\}$. Regarding time complexity, as discussed before, since the jumping join technique is still a specialized form of worst-case optimal join, its time complexity is $O(|E(L_E^Q)|^{\lfloor \frac{n}{2} \rfloor + 1})$, as discussed in Theorem 2.

Difference from Hash Join. Although the jumping worst-case optimal join technique looks like the hash join and also utilizes the hash table, it is different from existing hash join techniques. The hash join techniques is to join two relations sharing some common variables and build up the hash table using the matches of the join variables. However, in the jumping worst-case optimal join technique, there is indeed no common variables between $P^k(L_E^Q)$ and $P^{n-k-1}(L_E^Q)$. Our join technique is to integrate the extension operator with the intersection operator using a hash table.

V. Cost Model & Execution Plan

In this section, we discuss how to model the cost of different join techniques and propose a dynamic programming optimization to find the optimal execution plan. Note that, in this paper, our primary focus lies in extending existing worst-case optimal join techniques. Thus, our discussions are just about integrating the proposed jumping worst-case optimal join with other worst-case optimal join techniques. For the combination of our jumping worst-case optimal join and other join techniques, we can still use some previous methods [21].

A. Cost Model

Learning from the techniques in relational database systems [24], we can use simple formulas for estimating the cardinalities of the results of the join operator, based on its selectivity. The *selectivity factor* of an operator, that is, the proportion of tuples of an operand relation that participate in the result of that operation, is denoted by SF. It is a real value between 0 and 1, where a low value corresponds to a good (or high) selectivity and a high value to a bad (or low) selectivity. Thus, the cardinality of join is as follows.

$$card(R \bowtie S) = SF \times card(R) \times card(S) \qquad (2)$$

Then, the key issue to formulate the cost models of generic join and our jumping worst-case optimal join technique is how to estimate their selectivity factors. Based on Equation 2, the formula for estimating the selectivity factors is as follows.

$$SF = \frac{card(R \bowtie S)}{card(R) \times card(S)} \qquad (3)$$

Here, we first use the basic situation of generic join and our jumping worst-case optimal join technique to estimate the selectivity factors. For generic join, the basic situation is to extend the match of $P^1(L_E^Q)$ to $P^2(L_E^Q)$. For our jumping join technique, because a path query $P^n(L_E^Q)$ should be divided into two subqueries $P^k(L_E^Q)$ and $P^{n-k-1}(L_E^Q)$ ($k \leq 1$), the basic situation is $P^3(L_E^Q)$ which can be divided into two subqueries of $P^1(L_E^Q)$. For $P^2(L_E^Q)$ and $P^3(L_E^Q)$, the numbers of query results are both $|E(L_E^Q)|^2$ as discussed in Theorem 2, so the divisors in the formula for estimating their selectivity factors are $|E(L_E^Q)|^2$.

For generic join, if a partial match of $P^1(L_E^Q)$ can be extended to the match of $P^2(L_E^Q)$, the last vertex in the partial match should be a vertex that is the heads of another edges

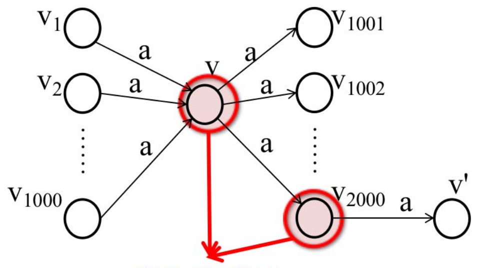


Fig. 4. Example Hub Vertices

in $E(L_E^Q)$. Meanwhile, it is obvious that the last vertex in the partial match is the tails of some edges in $E(L_E^Q)$. This indicates that the cardinality of $P^2(L_E^Q)$ highly depends on the set of vertices which can be both heads of some edges and tails of some other edges. We call the vertices *hub vertices* and denote them as $V_H(L_E^Q)$.

$$V_H(L_E^Q) = \{v \mid \exists w_1, w_1 \rightarrow v \in E \wedge \exists w_2, v \rightarrow w_2 \in E(G[L_E^Q])\}$$

For example, we highlight the two hub vertices v and v_{2000} in our extreme example graph in Fig. 4. v is tail of edges from $v_1, ..., v_{1000}$ and head of edges to $v_{1001}, ..., v_{2000}$, while v_{2000} is tail of edge from v and head of edges to v'.

The above discussion concerning hub vertices can also be extended to queries of other lengths. If a match of $P^k(L_E^Q)$ can be extended to a match of $P^{k+1}(L_E^Q)$, the final vertex in the match of $P^k(L_E^Q)$ should be a hub vertex, serving as the heads of other edges in $E(L_E^Q)$. Consequently, the distribution of edges adjacent to hub vertices becomes crucial for generic join to extend matches of $P^k(L_E^Q)$ to matches of $P^{k+1}(L_E^Q)$.

Here, we assume that the distribution of edges adjacent to hub vertices is uniform, and all edges are independent, meaning that the edges adjacent to one hub vertex does not affect other edges adjacent to any hub vertex. We use $N^-(V_H(L_E^Q))$ and $N^+(V_H(L_E^Q))$ to denote the in-edges and out-edges of hub vertices. Then, the number of matches of $P^2(L_E^Q)$ that can be extended from matches of $P^2(L_E^Q)$ is $\frac{(|N^+(V_H(L_E^Q))| \times |N^-(V_H(L_E^Q))|)}{|V_H(L_E^Q)|}$.

Hence, we can estimate the selectivity factor SF_{GJ} for generic join as follows.

$$SF_{GJ} = \frac{(|N^+(V_H(L_E^Q))| \times |N^-(V_H(L_E^Q))|)/|V_H(L_E^Q)|}{|E(L_E^Q)|^2} \qquad (4)$$

In contrast, for our jumping worst-case optimal join technique, if a partial match of $P^1(L_E^Q)$ can be extended and intersected with the partial match of $P^1(L_E^Q)$, the last vertex in the partial match of $P^1(L_E^Q)$ should be a vertex that is the heads of another edges in $E(L_E^Q)$ and the first vertex in the partial match of $P^1(L_E^Q)$ should be the tails of some edges in $E(L_E^Q)$. This indicates that the join cost of our technique highly depends on the set of edges where their heads are tails of some other edges or their tails are heads of some other edges, which is denoted as $E_H(L_E^Q)$.

$$E_H(L_E^Q) = \{u \rightarrow v \mid \exists w_1, w_1 \rightarrow u \in E \wedge \exists w_2, v \rightarrow w_2 \in E(L_E^Q)\}$$

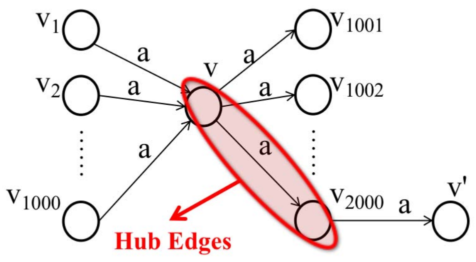


Fig. 5. Example Hub Edges

For example, we highlight the hub edge $v \xrightarrow{\{a\}} v_{2000}$ in our extreme example graph in Fig. 5. v is tail of edges from $v_1, ..., v_{1000}$ and v_{2000} is head of edges to v'.

The above discussion concerning hub edges can also be extended to queries of other lengths. For a variable-length path query $P^n(L_E^Q)$, if a match of $P^k(L_E^Q)$ $(k < n-1)$ can be extended and intersected with the match of $P^{n-k-1}(L_E^Q)$, the final vertex in the match of $P^k(L_E^Q)$ should be connected to the first vertex in the match of $P^{n-k-1}(L_E^Q)$ via a hub edge. Consequently, the distribution of edges adjacent to hub edges becomes crucial for jumping worst-case optimal join to join matches of $P^k(L_E^Q)$ with matches of $P^{n-k-1}(L_E^Q)$.

Similar to generic join, we also assume that the distribution of edges adjacent to hub edges is uniform, and all edges are independent, meaning that the edges adjacent to one hub edge does not affect other edges adjacent to any hub edge. We use $N^-(E_H(L_E^Q))$ and $N^+(E_H(L_E^Q))$ to denote the in-edges of hub edges' heads and out-edges of hub edges' tails. Then, the number of matches of $P^3(L_E^Q)$ that can be generated by jumping worst-case optimal joins of matches of $P^1(L_E^Q)$ is $\frac{(|N^+(E_H(L_E^Q))| \times |N^-(E_H(L_E^Q))|)}{|E_H(L_E^Q)|}$.

Thus, we can estimate the selectivity factor SF_{J-WCO} for our jumping worst-case optimal join technique as follows.

$$SF_{J-WCO} = \frac{(|N^+(E_H(L_E^Q))| \times |N^-(E_H(L_E^Q))|)/|E_H(L_E^Q)|}{|E(L_E^Q)|^2} \tag{5}$$

To estimate the selectivity factors SF_{GJ} and SF_{J-WCO}, we need to compute the numbers of edges adjacent to hub vertices and edges of each edge label. This can be computed in both online and offline. For both generic join and our jumping join, the first step is to evaluate the path query of a single query edge with a SCAN operator. It is easy for us to compute the edges adjacent to hub vertices and edges by scanning the results of the SCAN operator once. We can also compute and maintain the selectivity factors for each edge label offline. Given a set of edges with labels in L_E^Q, since we just need to scan the edges to compute the selectivity factors, the time complexity is $O(|E(L_E^Q)|)$.

B. Dynamic Programming Query Plan Generation

For each n-edge path query $P^n(L_E^Q)$, its lowest cost query plan can be in two different ways:

- It extends $P^{n-1}(L_E^Q)$ through the Extension/Intersection operator in Generic Join;

$$\begin{tikzcd} P^1(\{a\}) \arrow[r, shift left, "\text{J-WCO}"] \arrow[r, shift right, "\text{J-WCO}"'] & P^3(\{a\}) \end{tikzcd} \qquad \begin{tikzcd} P^1(\{a\}) \arrow[r, "\text{GJ}"] & P^2(\{a\}) \arrow[r, "\text{GJ}"] & P^3(\{a\}) \end{tikzcd}$$

(a) (b)

Fig. 6. Query Plans for $P^3(\{a\})$

- It merges $P^k(L_E^Q)$ and $P^{n-k-1}(L_E^Q)$ $(1 \leq k \leq n-2)$ through our jumping worst-case optimal join technique;

Then, based on the selectivity factors of SF_{GJ} and SF_{J-WCO}, the costs of query plans generated in the above two ways can satisfy the following recurrence.

$$\begin{aligned} &card(P^n(L_E^Q)) = \\ &\min_{1 \leq k \leq n-2}\{SF_{GJ} \times card(P^{n-1}(L_E^Q)) \times card(P^1(L_E^Q)), \\ &SF_{J-WCO} \times card(P^k(L_E^Q)) \times card(P^{n-k-1}(L_E^Q))\} \end{aligned} \tag{6}$$

Here, a query plan of $P^n(L_E^Q)$ can be represented as a directed acyclic graph (DAG), denoted as $QP(P^n(L_E^Q))$. Each vertex v_p in $QP(P^n(L_E^Q))$ corresponds to the plan of path query $P^k(L_E^Q)$ $(1 \leq k \leq n)$, and there are at most two edges pointing to v_p. If a path query $P^k(L_E^Q)$ is extended to $P^{k+1}(L_E^Q)$ through Generic Join, the vertex of $P^k(L_E^Q)$ has an edge labelled with Generic Join (GJ) to the vertex of $P^{k+1}(L_E^Q)$; if two path queries $P^{k_1}(L_E^Q)$ and $P^{k_2}(L_E^Q)$ join together through our join technique, their vertices have edges labelled with our jumping worst-case optimal join technique (J-WCO) to the vertex of $P^{k_1+k_2+1}(L_E^Q)$. For example, there are two possible query plans for $P^3(\{a\})$ and their corresponding DAGs are shown in Fig. 6.

It is obvious that there is only one vertex that has no incoming edges from other vertices in the DAG for a query plan and corresponds to $P^1(L_E^Q)$. $P^1(L_E^Q)$ is computed through the Scan operator but not the extension or join operator. For a path query $P^n(L_E^Q)$ of a single constraint, there is also only one vertex that has no outgoing edges and corresponds to $P^n(L_E^Q)$.

Based on the recurrence in Equation 6, we can design a dynamic programming algorithm that evaluates the recurrence to find the optimal query plan, as described in Algorithm 2.

The input to the algorithm is a set of queries expressed as a path query of n edges, $P^n(L_E^Q)$. Initially, we create a map, $QPCMap$, to maintain the path query and its corresponding cardinality (Line 1 in Algorithm 2). The cardinality of $P^1(L_E^Q)$ can be obtained firstly by using a SCAN operator in both generic join and our jumping join, so it is directly added into $QPCMap$ (Line 2 in Algorithm 2). We also initialize the query plan of $QP(P^i(L_E^Q))$ with a vertex corresponding to $P^1(L_E^Q)$ (Line 3 in Algorithm 2). Then, starting from $i = 3$ up to n, for each path query of i edges, we find the lowest cost plan to compute $P^i(L_E^Q)$ in two different ways (Lines 4-10 in Algorithm 2): the query plan of $P^i(L_E^Q)$ is extended from the results of $P^i(L_E^Q)$ by the E/I operator in generic join; or the query plan of $P^i(L_E^Q)$ is get from the results of $P^k(L_E^Q)$ and $P^{i-k-1}(L_E^Q)$ by our jumping join technique. The cardinality of $P^i(L_E^Q)$ is also added into $QPCMap$ (Line 11 in Algorithm 2). Last, we generate the DAG of the optimal

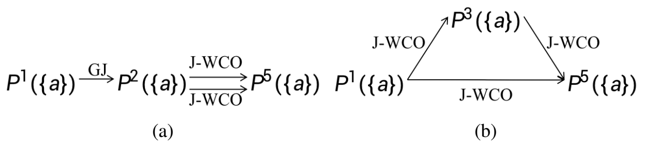


Fig. 7. Query Plans for $P^5(\{a\})$

query plan according to the way of computing the lowest cost plan (Lines 12-16 in Algorithm 2).

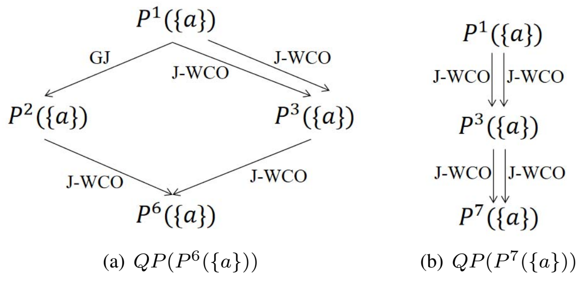
Algorithm 2: Dynamic Programming Query Plan Generation

```
Input: A Path Query of $n$ Edges, $P^n(L_E^Q)$
Output: The Optimal Query Plan $QP(P^n(L_E^Q))$
1  Initialize a map $QPCMap$;
2  Add $P^1(L_E^Q)$ and its cardinality in $QPCMap$;
3  Initialize a vertex in $QP(P^n(L_E^Q))$ corresponding to
     $P^1(L_E^Q)$;
4  for $i = 2$ to $n$ do
5      $minC \leftarrow SF_{GJ} \times QPCMap(P^{i-1}(L_E^Q)) \times$
         $QPCMap(P^1(L_E^Q))$;
6      $k \leftarrow 0$;
7      for $j = 1$ to $i - 1$ do
8          if $minC > SF_{J-WCO} \times QPCMap(P^j(L_E^Q)) \times$
             $QPCMap(P^{i-j-1}(L_E^Q))$ then
9              $minC \leftarrow SF_{J-WCO} \times QPCMap(P^j(L_E^Q) \times$
                 $QPCMap(P^{i-j-1}(L_E^Q)))$;
10             $k \leftarrow j$;
11     $QPCMap(P^i(L_E^Q) \leftarrow minC$;
12     Initialize a vertex in $QP(P^n(L_E^Q))$ corresponding to
         $P^i(L_E^Q)$;
13     if $k = 0$ then
14         Add an edge with label Generic Join in
             $QP(P^n(L_E^Q))$ from $P^{i-1}(L_E^Q)$ to $P^i(L_E^Q)$;
15     else
16         Add two edges with label jumping Join in
             $QP(P^n(L_E^Q))$ from $P^k(L_E^Q)$ and $P^{i-k-1}(L_E^Q)$ to
             $P^i(L_E^Q)$;
17 Remove the vertices without outgoing edges in
     $QP(P^n(L_E^Q))$ except the vertex of $P^n(L_E^Q)$;
18 Return $QP(P^n(L_E^Q))$;
```

Here, we should note that the above algorithm does not guarantee the worst-case optimality. For example, if the query plan for $P^5(\{a\})$ in Fig. 7(a) is selected where the results of two $P^2(\{a\})$ are joined through our jumping worst-case optimal join technique, then the number of results for $P^5(\{a\})$ becomes at most $O(|E(L_E^Q)|^5)$. This is because the number of results for $P^2(\{a\})$ can be at most $O(|E(L_E^Q)|^2)$ and the limit of SF_{J-WCO} can be 1. However, this query plan probably cannot be selected as the optimal plan, if the number of results for $P^2(\{a\})$ is so large. We can select the query plan in Fig. 7(b) as the optimal one, which can guarantee the worst-case optimality.

1) Extension to Path Query with Range Constraint: Here, we discuss how to extend our plan generation method to generate the query plan of a path query with range constraints $P^{[m,n]}(L_E^Q)$.

Given a path query $P^{[m,n]}(L_E^Q)$, it can be deemed as $m - n + 1$ path queries from $P^m(L_E^Q)$ to $P^n(L_E^Q)$. In our dynamic programming query plan generation algorithm (Algorithm 2), when we compute the optimal query plan for $P^n(L_E^Q)$, we enumerate and maintain the optimal query plan for path query of any length k ($1 \leq k \leq n$). Thus, the query plans from $P^m(L_E^Q)$ to $P^n(L_E^Q)$ can be generated after our query plan generation. We can merge these query plans together to get the query plan of $P^{[m,n]}(L_E^Q)$.



(a) $QP(P^6(\{a\}))$ (b) $QP(P^7(\{a\}))$

Fig. 8. Query Plans $QP(P^6(\{a\}))$ and $QP(P^7(\{a\}))$

Then, different from the query plan for a path query of single constraint, there may be multiple vertices that have no outgoing edges and correspond to the query plans from $P^m(L_E^Q)$ to $P^n(L_E^Q)$.

For example, let us consider $P^{[6,7]}(\{a\})$. We can directly use our query plan generation algorithm (Algorithm 2) to find out the query plans of both $P^6(\{a\})$ and $P^7(\{a\})$ during generating the query plan of $P^7(\{a\})$. Here, we assume that the query plans for $P^6(\{a\})$ and $P^7(\{a\})$ are shown in Figures 8(a) and 8(b). Then, they can be merged with the plan of $P^{[6,7]}\{a\})$ as shown in Fig. 9.

VI. Parallelism

To further improve the performance of path query evaluation, we discuss how to exploit the intra-query parallelism within the query evaluation, which can break a single path query plan into some pieces and execute them in parallel.

For a path query, its results can be computed by joining two path queries of smaller lengths via our jumping-like worst-case optimal join technique. If the two path queries of smaller lengths are independent on each other, they can be evaluated in parallel. For example, for the query plan for $QP(P^6(\{a\}))$ in Figure 8(a), $P^6(\{a\})$ are computed by joining the results of $P^2(\{a\})$ and $P^3(\{a\})$, while $P^2(\{a\})$ and $P^3(\{a\})$ can be evaluated independently. Thus, we can first compute the results of $P^2(\{a\})$ and $P^3(\{a\})$ in parallel, and then join their results to obtain the final results.

Concretely, given the DAG $QP(P^{[m,n]}(L_E^Q))$ of the query plan for the path query $P^{[m,n]}(L_E^Q)$, if there is no a path from $P^i(L_E^Q)$ to $P^j(L_E^Q)$ and $P^j(L_E^Q)$ to $P^i(L_E^Q)$, then $P^i(L_E^Q)$ and $P^j(L_E^Q)$ can be evaluated independently. Thus, we can find all independent path queries of smaller lengths in $QP(P^{[m,n]}(L_E^Q))$ to schedule and parallelize the evaluation of $P^{[m,n]}(L_E^Q)$.

In this paper, we propose a parallel query execution algorithm in Algorithm 3, which parallelize the queries in $QP(P^{[m,n]}(L_E^Q))$ layer by layer. The algorithm is an iterative

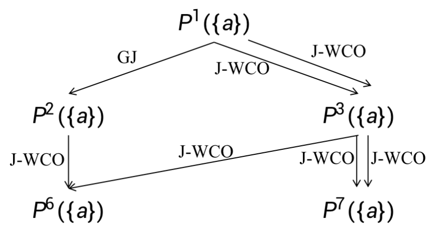


Fig. 9. Query Plan $QP(P^{[6,7]}(\{a\}))$

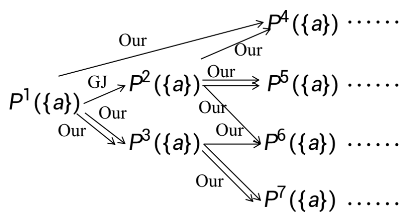


Fig. 10. Query Plan $QP(P^{[1,+\infty]}(\{a\}))$

algorithm. At each iteration, we find a set S of path queries in $QP(P^{[m,n]}(L_E^Q))$ which have no incoming edges (Lines 3-4 in Algorithm 3). Then, we evaluate these path queries of smaller lengths in parallel (Line 5 in Algorithm 3), and remove these queries and all its outgoing edges in $QP(P^{[m,n]}(L_E^Q))$ (Line 6-7 in Algorithm 3).

Algorithm 3: Parallel Query Execution

```
   Input: $QP(P^{[m,n]}(L_E^Q))$
   Output: $[\![P^{[m,n]}(L_E^Q)]\!]$
1  $S \leftarrow \emptyset$;
2  while $QP(P^{[m,n]}(L_E^Q))$ is not empty do
3      for each vertex $P^i(L_E^Q)$ without incoming edges in $QP(P^{[m,n]}(L_E^Q))$ do
4          Add $P^i(L_E^Q)$ into $S$;
5      Execute all path queries in $S$ in parallel;
6      for each vertex $P^i(L_E^Q)$ in $S$ do
7          Remove $P^i(L_E^Q)$ and all its outgoing edges in $QP(P^{[m,n]}(L_E^Q))$;
8          if $m \le i \le n$ then
9              Add $[\![P^i(L_E^Q)]\!]$ into $[\![P^{[m,n]}(L_E^Q)]\!]$;
10     $S \leftarrow \emptyset$;
11 Return $[\![P^{[m,n]}(L_E^Q)]\!]$;
```

For example, let us consider the query plan of $QP(P^{[6,7]}(\{a\}))$ in Figure 9. At the first iteration, the only one vertex without incoming edge in $QP(P^{[6,7]}(\{a\}))$ is $P^1(\{a\})$. It is added to S for execution and all its outgoing edges are removed. Then, at the second iteration, there are two vertices without incoming edge $P^2(\{a\})$ and $P^3(\{a\})$. These two queries are added into S and executed in parallel. After $P^2(\{a\})$ and $P^3(\{a\})$ are removed in $QP(P^{[6,7]}(\{a\}))$, there are two remaining vertices without incoming edge $P^6(\{a\})$ and $P^7(\{a\})$. They are executed in parallel, and the results are added to the set of final results to return.

VII. Extension for Path Queries without Length Limit

In real query logs, we find that a more widely used kind of query is the query with a wildcard, which corresponds to the path query without the length limit, i.e. $P^{[1,+\infty]}(L_E^Q)$. Since the kind of query do not limit the length, paths of any lengths between two variables should be returned. Thus, we can incrementally generate the query plans from 1 to infinity and merge these plans together. Note that, the query plan generation and query execution can be parallelized. If the results of all subqueries used to be joined to the results of a query have been found out, we directly initialize a thread to execute the join. Figure 10 shows a query plan for the path query without the length limit.

VIII. System Implementation

We implement our new techniques on top of a property graph system, Graphflow [16], and an RDF graph system, gStore [30] . We release our extensions on both Graphflow[1] and gStore[2].

Graphflow is a single machine, multi-threaded, main memory graph DBMS implemented in Java. The system supports a subset of the Cypher language [12]. Graphflow indexes both the forward and backward adjacency lists and store them in sorted vertex ID order. Although Cypher supports variable-length path queries, the original version of Graphflow does not support variable-length path queries. We seamlessly integrate our proposed methods into Graphflow by implementing a jumping worst-case optimal join operator. Specifically, when handling a variable-length path query, we initially utilize Graphflow's Scan function to fetch edges that satisfy the label constraints. Then, leveraging the retrieved edges, we calculate the selectivity factors of hub vertices and edges, followed by employing our dynamic programming algorithm to generate the query plan. Finally, guided by the query plan, we use both Graphflow's worst-case optimal join operator and our jumping worst-case optimal join operator to obtain the final results.

gStore is a disk-based RDF graph system which stores and retrieves RDF graphs from a graph database perspective. It follows the "filter-and-join" framework. In the offline phase, it first designs a vertex signature encoding method and build up an index named VS*-tree. In online phase, gStore first use the index to filter the candidates of each query vertex and then join them together to form the final results. As well, although SPARQL 1.1 supports variable-length path queries, the original version of gStore does not support variable-length queries. We embed our proposed techniques during joining the candidates of query vertices. In gStore, we implement a class named JumpingLikeJoin for our jumping worst-case optimal join technique. Initially, we retrieve the edges with the specified labels to initialize the JumpingLikeJoin class, utilizing the KV-store interface provided by gStore. Then, we calculate the selectivity factors of hub vertices and edges and formulate the query plan accordingly. Lastly, we implement the execution of the query plan to obtain the final results.

IX. Experimental Results

We evaluate our approach on a range of different graph datasets, including both synthetic and real-world graphs. These

[1]https://github.com/15197580192/jumping-wco-join-graphflow
[2]https://github.com/15197580192/jumping-wco-join/

TABLE I
Statistics of Property Graphs

Dataset	$\|V\|$	$\|E\|$	$\|L\|$
LDBC 2M	586,530	2,050,261	20
LDBC 12M	1,844,718	12,399,300	20
LDBC 26M	4,010,012	26,319,840	20
LiveJournal	4,847,571	68,993,773	80

TABLE II
Statistics of RDF Graphs

Dataset	$\|V\|$	$\|E\|$	$\|L\|$
DBpedia 100M	19,027,155	120,336,646	52,603
DBpedia 500M	65,588,189	512,213,528	73,444
DBpedia 1B	139,493,254	1,111,481,066	124,034
YAGO2	21,073,153	284,417,966	98

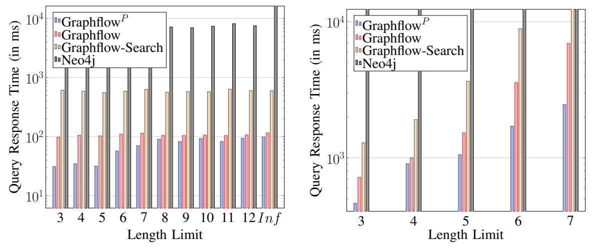
datasets can be further classified into two models: property graphs and RDF graphs. Tables I and II present the statistics for these graph datasets.

For property graphs, we utilize a well-established benchmark, LDBC [7], to generate three datasets of edges ranging from 2 million to 20 million for assessing the scalability of the system. Additionally, we download a real-world property graph LiveJournal, through SNAP [19], which lacks edge labels. We randomly assign edge labels to them and vary the number of labels from 20 to 100, in increments of 20, to study the impact of the number of labels. By default, the number of labels is 80.

For RDF graphs, we utilize DBpedia [18] to get three datasets for evaluating the scalability of the system. DBpedia is a project that extracts RDF data from Wikipedia, and multiple versions of DBpedia are available, regularly updated to reflect changes in Wikipedia content. For our experiments, we use three versions of DBpedia containing 100 million, 500 million, and 1 billion triples, denoted as DBpedia 100M, DBpedia 500M, and DBpedia 1B, respectively. DBpedia 100M is derived from DBpedia version 3.4, DBpedia 500M is from version 3.8, and DBpedia 1B is from version 2014. By default, we use the DBpedia 100M dataset. Additionally, we test our methods on another real RDF graph dataset, YAGO2 [13].

For LiveJournal, we randomly select edge labels to construct 5 queries. However, for LDBC, DBpedia, and YAGO2, where edge labels are not randomly assigned and only a limited number of edge labels exhibit transitivity, we are only able to formulate one transitive edge label query for each dataset. Subsequently, we execute each query multiple times and report the average query response time. Specifically, for LDBC, we employ one of the LDBC benchmark queries named IS2 as a template, systematically generating queries with length limits; for DBpedia, we use the property $KNOWS$; for DBpedia, we use the property $influencedBy$; for YAGO2, our selection is the property $hasInternalWikipediaLinkTo$.

Our extensions of gStore and Graphflow are denoted as Graphflow^P and gStore^P, of which the implementation details are in Section VIII. For property graphs, we compare Graphflow^P with the most well-known disk-based property graph system, Neo4j [2], and the original version of Graphflow [16]. For RDF graphs, we compare gStore^P with a disk-based RDF graph system supporting the worst-case optimal join, Jena-LFJ [14], and the original version of gStore [30]. We also implement a baseline that processes a variable-length path query by traversing the graph from the query vertex candidates, denoted as Graphflow-Search and gStore-Search. Further, when evaluating our proposed parallel optimization, we add a baseline processing a variable-length path query by executing each length as a join query and then taking the union, denoted as Graphflow-UNION and gStore-UNION.



(a) LDBC (b) LiveJournal

Fig. 11. Efficiency Test of the Proposed Techniques on Property Graphs

All experiments are conducted on a single core of an CentOS 8.4 computer, with 32 CPUs, 313GB RAM and 2TB magnetic disk.

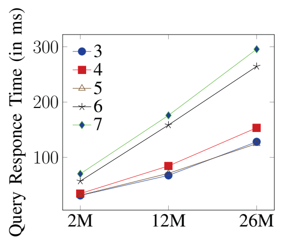
A. Experiments on Property Graphs

1) Efficiency Test: In this experiment, we assess the effectiveness of the proposed techniques on different property graphs, using queries with length limits from 3 to infinity. Fig. 11 shows the results of the experiments conducted on both LDBC and LiveJournal. Note that, queries with length limits exceeding 7 result in an excessive number of results in LiveJournal, causing query response times to surpass the time limit. As a result, we only present the results for queries with length limits exceeding 7 in LDBC.

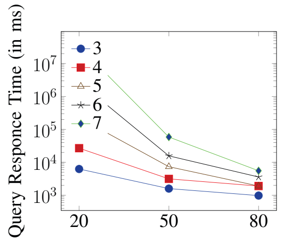
In general, Graphflow^P consistently demonstrates a capacity to reduce query response times across different graphs when compared to the other three systems. This efficacy is primarily attributed to the reduction in the number of intermediate results facilitated by our jumping worst-case optimal technique. Particularly noteworthy is the observation that, for variable-length path queries with lengths of 6 and 7 on LiveJournal, Graphflow^P can outperform Graphflow by a factor of two.

2) Scalability Test: In this section, we employ the LDBC dataset to assess the scalability of our method by varying the dataset sizes. Additionally, we leverage the LiveJournal dataset to evaluate scalability by varying the number of labels in both data graphs and queries, since only the labels in the LiveJournal dataset are randomly assigned. Last, we also use LiveJournal to assess performance while varying the number of cores for parallelism.

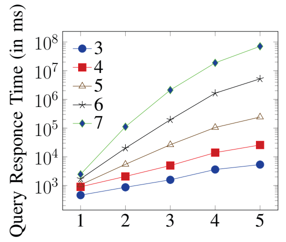
a) Varying Graph Size: In this experiment, we utilize LDBC to examine the influence of dataset size on our techniques. We systematically explore the impact of varying dataset sizes, generating three LDBC graphs with edges ranging from 2 million to 26 million. The outcomes of this analysis, varying the number of length limits from 3 to 7, are presented in Fig. 12(a).



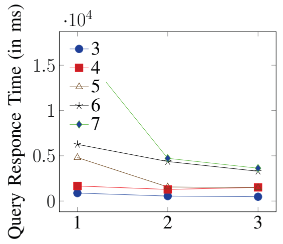
(a) Varying Size of Graphs



(b) Varying Number of Labels on Graphs



(c) Varying Number of Labels on Queries



(d) Varying Number of Cores

Fig. 12. Scalability Test on Property Graphs

Broadly, as the dataset size increases, there is a discernible rise in the number of intermediate results, leading to an associated increase in query response times. This trend is consistent with the expected behavior of systems handling larger datasets, highlighting the importance of scalability considerations in the deployment of our techniques.

b) Varying Number of Labels on Graphs: In this experiment, we analyze the influence of the number of labels using LiveJournal, given its randomly assigned labels. We systematically vary the number of labels from 20 to 80, in increments of 30, and present the outcomes in Fig. 12(b).

Broadly speaking, a reduced number of labels on graphs implies that a query edge maps to a larger pool of candidates. Consequently, the same variable-length path query becomes more selective within a graph characterized by fewer labels. This heightened selectivity leads to a decrease in the number of matches, resulting in fewer computations and, consequently, improved performance for the proposed method.

c) Varying Number of Labels on Queries: In this experiment, we use LiveJournal to test the influence of the number of labels on queries, given its randomly assigned labels. We vary the number of labels from 1 to 5 and test queries of the length limit from 3 to 7. The results are shown in Fig. 12(c).

Generally speaking, with an increase in the number of labels on queries, each query edge tends to map to a larger pool of candidates. Thus, the variable-length path query becomes less selective, leading to a higher number of matches. This tends to result in longer query response times.

d) Varying Number of Cores: In this experiment, we also use LiveJournal to evaluate performance while varying the number of cores for parallelism. We explore a range of 1 to 3 cores with each core corresponding to one thread and test queries with length limits from 3 to 7. The results are depicted in Fig. 12(d).

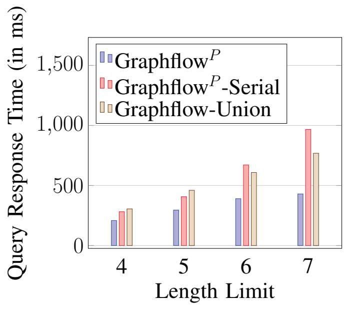


Fig. 13. Efficiency Test of Parallelism on Property Graphs

Generally speaking, as the number of cores increases, so does the level of parallelism in evaluating a variable-length path query. This effect is particularly pronounced due to our parallelism optimization, which enables intra-query parallelism by processing different segments of the query simultaneously. Thus, a larger number of cores tends to yield shorter query response times.

3) Evaluation of Our Proposed Optimizations: In this experiment, we employ the LiveJournal dataset to investigate the impact of the parallelism optimization discussed in Section VI. The results of this experiment are presented in Fig. 13. Two baselines are compared, where the first one denoted as Graphflow$P$-Serial does not our optimization of parallelism and the second one denoted as Graphflow-UNION processes a variable-length path query by executing each length as a join query and then taking the union.

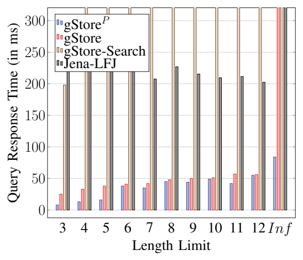
It is noteworthy that the parallelism technique outlined in Section VI can only be employed when the length limit for a variable-length path query is not smaller than 4. Therefore, in this experiment, we specifically vary the length limit of queries from 4 to 7. As depicted in Fig. 13, the optimization proposed in Section VI consistently enhances query performance compared to Graphflow$P$-Serial and Graphflow-UNION.

B. Experiments on RDF Graphs

1) Efficiency Test: In this experiment, we assess the optimization techniques introduced in the paper on RDF graphs. We also systematically vary the length limit for queries from 3 to infinity. Fig. 14 shows the results on different graphs. Note that, queries with length limits exceeding 7 yield an excessive number of results in YAGO2, causing query response times to exceed the designated time limit. As a result, we solely report the response times of queries with length limits exceeding 7 in DBpedia.

Analogous to the findings in the property graph experiment, gStore^P consistently outperforms gStore and gStore-Search when leveraging our techniques. Furthermore, gStore^P demonstrates competitive efficiency with Jena-LFJ, where Jena-LFJ supports worst-case optimal joins. This experiment underscores the significant performance enhancement achieved by the optimized gStore^P system for variable-length path queries.

Notably, in YAGO2, when the query length is between 3 and 5, gStore^P exhibits a remarkable performance improvement,


(a) DBpedia

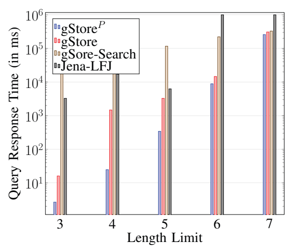

(b) YAGO2

Fig. 14. Efficiency Test of the Proposed Techniques on RDF Graphs

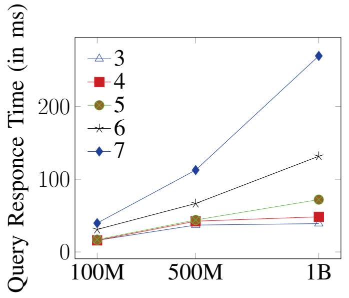


Fig. 15. Scalability Test of the Proposed Techniques on RDF Graphs

surpassing gStore by nearly 80% and outperforming Jena-LFJ by one order of magnitude. This noteworthy enhancement is attributed to the inherent selectivity of real RDF graphs like DBpedia and YAGO2. As the path length increases, the number of query results tends to decrease in most real RDF graphs. A variable-length path query with a smaller length limit maps to a larger set of results, leading to an augmented presence of irrelevant intermediate results. Our method efficiently circumvents the computationally intensive matching process for these irrelevant results, contributing to the observed performance gains.

2) Scalability Test: In this section, we utilize the DBpedia dataset to evaluate the scalability of our method by systematically varying the dataset sizes. Notably, in real RDF graphs and their logs such as DBpedia and YAGO2, properties are not randomly assigned. Consequently, we do not conduct experiments involving the variation of the number of properties.

a) Varying Graph Size: In this experiment, we assess the impact of dataset size on our techniques within the domain of RDF graphs. Using three DBpedia graphs with sizes from 10 million to 100 million edges, we study the scalability of our method under varying dataset sizes. The results, considering length limits ranging from 3 to 7, are depicted in Fig. 15.

The observed linear increase in query time across the three datasets indicates that the method proposed in this paper adeptly accommodates dataset expansion within the same dataset type. Consequently, we deduce that the proposed method consistently upholds robust query performance across different scales of datasets belonging to the same type. This finding underscores the scalability and adaptability of our techniques in the context of RDF graphs.

3) Evaluation of Our Proposed Optimizations: In this experiment, we employ the DBpedia 100M dataset to evaluate the impact of the parallelism optimization discussed in Section VI on RDF graphs. There are also two baselines for comparison, The first one denoted as gStore$P$-Serial does not our optimization of parallelism and the second one denoted as gStore-UNION processes the query by executing each length as a join query and then taking the union.

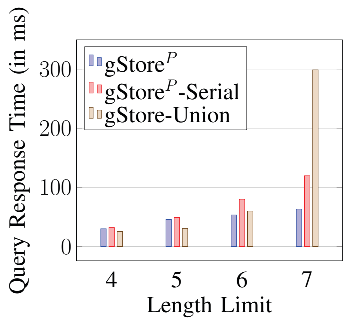


Fig. 16. Efficiency Test of Parallelism on RDF Graphs

The outcomes, depicted in Fig. 13, are similar to those observed in property graphs. The optimization proposed in Section VI consistently enhances query performance across both RDF and property graphs.

X. Conclusion and Future Work

In this paper, we present a novel worst-case optimal join technique, called jumping worst-case optimal join, for processing variable-length path queries. To integrate this proposed join technique with existing join techniques, we develop a cost model that enables us to devise a dynamic programming optimization approach for finding the most efficient query plan. Moreover, we introduce several optimizations to leverage intra-query parallelism during the evaluation process. To show the effectiveness of our techniques, we implement them in two graph systems: Graphflow, a property graph system, and gStore, an RDF graph system. Through extensive experiments, we verify the superiority of our proposed techniques.

While jumping worst-case optimal join can be extended to handle general queries or even queries on hybrid-type databases, its superiority for them may not be as high as in variable-length path queries. This extension necessitates the presence of a substructure occurring multiple times within the query. Then, this substructure is executed once and jumping worst-case optimal join is employed to rapidly join the results. In a variable-length path query, the substructure is typically evident as an edge. However, in general queries or queries on hybrid-type databases, the repeated occurrence of such a substructure may not be guaranteed. Thus, the extension of this technique requires further study.

Acknowledgment

The corresponding author is Peng Peng. This work was supported by the National Key R&D Projects (2022YFB3103500), Science and Technology Major Projects of Changsha City (No. kh2205032), NSFC under grants (U22A2030,U20A20174,61932001), Technology Projects of Hunan Province (2015TP1004) and Hunan Provincial Natural Science Foundation of China under grant (2022JJ30165).

REFERENCES

[1] Apache Jena. https://jena.apache.org/, 2020.

[2] Neo4j. https://neo4j.com/, 2020.

[3] RDF4J. https://rdf4j.org/, 2020.

[4] Virtuoso. https://virtuoso.openlinksw.com/, 2020.

[5] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.*, 42(4), Oct. 2017.

[6] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, SIGMOD '16, page 431–446, New York, NY, USA, 2016. Association for Computing Machinery.

[7] R. Angles, J. B. Antal, A. Averbuch, P. A. Boncz, O. Erling, A. Gubichev, V. Haprian, M. Kaufmann, J. Larriba-Pey, N. Martínez-Bazan, J. Marton, M. Paradies, M. Pham, A. Prat-Pérez, M. Spasic, B. A. Steer, G. Szárnyas, and J. Waudby. The LDBC Social Network Benchmark. *CoRR*, abs/2001.02299, 2020.

[8] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[9] Y. Cai and W. Zheng. Answering label-constrained reachability queries via reduction techniques. In *Database Systems for Advanced Applications*, pages 114–131, Cham, 2023. Springer Nature Switzerland.

[10] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. TigerGraph: A Native MPP Graph Database. *CoRR*, abs/1901.08248, 2019.

[11] W. Fan, T. He, L. Lai, X. Li, Y. Li, Z. Li, Z. Qian, C. Tian, L. Wang, J. Xu, Y. Yao, Q. Yin, W. Yu, K. Zeng, K. Zhao, J. Zhou, D. Zhu, and R. Zhu. Graphscope: A Unified Engine For Big Graph Processing. *Proc. VLDB Endow.*, 14(12):2879–2892, 2021.

[12] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*, pages 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[13] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In *WWW*, page 229–232, New York, NY, USA, 2011. Association for Computing Machinery.

[14] A. Hogan, C. Riveros, C. Rojas, and A. Soto. A Worst-Case Optimal Join Algorithm for SPARQL. In *ISWC*, pages 258–275, Cham, 2019. Springer International Publishing.

[15] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing Label-Constraint Reachability in Graph Databases. In *SIGMOD*, SIGMOD '10, page 123–134, New York, NY, USA, 2010. Association for Computing Machinery.

[16] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An Active Graph Database. In *SIGMOD*, pages 1695–1698, New York, NY, USA, 2017. Association for Computing Machinery.

[17] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, Y. Zhang, Z. Qian, and J. Zhou. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.*, 12(10):1099–1112, 2019.

[18] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. van Kleef, S. Auer, and C. Bizer. DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[19] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[20] H. Ma, B. Shao, Y. Xiao, L. J. Chen, and H. Wang. G-SQL: Fast Query Processing via Graph Exploration. *Proc. VLDB Endow.*, 9(12):900–911, aug 2016.

[21] A. Mhedhbi and S. Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB*, 12(11):1692–1704, 2019.

[22] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case Optimal Join Algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.

[23] H. Q. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record*, 42(4):5–16, 2013.

[24] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, Cham, 2020.

[25] Y. Peng, X. Lin, Y. Zhang, W. Zhang, and L. Qin. Answering reachability and k-reach queries on large graphs with label constraints. *VLDB J.*, 31(1):101–127, 2022.

[26] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang. Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond. *Proc. VLDB Endow.*, 13(6):812–825, Feb. 2020.

[27] L. D. Valstar, G. H. Fletcher, and Y. Yoshida. Landmark indexing for evaluation of label-constrained reachability queries. In *SIGMOD*, SIGMOD '17, page 345–358, New York, NY, USA, 2017. Association for Computing Machinery.

[28] K. Xu, L. Zou, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Answering Label-Constraint Reachability in Large Graphs. In *CIKM*, CIKM '11, page 1595–1600, New York, NY, USA, 2011. Association for Computing Machinery.

[29] Y. Zeng, W. Yang, X. Zhou, G. Xiao, Y. Gao, and K. Li. Distributed set label-constrained reachability queries over billion-scale graphs. In *ICDE*, pages 1969–1981. IEEE, 2022.

[30] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. gStore: a graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590, 2014.

[31] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Efficient Processing of Label-Constraint Reachability Queries in Large Graphs. *Inf. Syst.*, 40:47–66, Mar. 2014.