A Unified Query Planning Framework for Conjunctive Regular Path Queries

Yue Pang
Peking University & HKUST
China
michelle.py@pku.edu.cn

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

Angela Bonifati
Lyon1 University, CNRS Liris & IUF
France
angela.bonifati@univ-lyon1.fr

M. Tamer Özsu
University of Waterloo
Canada
tamer.ozsu@uwaterloo.ca

Xiaofang Zhou
HKUST
Hong Kong SAR, China
zxf@cse.ust.hk

ABSTRACT

Conjunctive regular path queries (CRPQs) form a critical backbone of modern graph query languages, integrating subgraph matching with regular path queries (RPQs). Despite their ubiquity in applications like social networks, finance, and scientific data analysis, CRPQ optimization lacks a unified framework, relying instead on heuristic combinations of disjoint techniques for subgraph matching and RPQs. This paper bridges this gap by introducing a novel algebraic optimization framework for CRPQs. We propose a hypergraph query model that enables composability, addressing a longstanding challenge in graph query languages including GQL and SQL/PGQ, and define six core operators (TI, KC, SJ, UNION, INV, and SCAN) to abstract CRPQ semantics. Leveraging algebraic transformation rules, we enumerate a rich space of equivalent query plans and devise a cost-based optimizer to select near-optimal plans for execution. Implemented based on MillenniumDB and Neo4j, our framework achieves significant speedups on CRPQs extracted from the WDBench and LDBC SNB benchmarks.

PVLDB Reference Format:
Yue Pang, Lei Zou, Angela Bonifati, M. Tamer Özsu, and Xiaofang Zhou. A Unified Query Planning Framework for Conjunctive Regular Path Queries. PVLDB, 19(10): 2686 - 2698, 2026.
doi:10.14778/3828612.3828624

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SoftlySpoken/CRPQ-unified-planning.

1 INTRODUCTION

The recent advent of graph data in various application scenarios, including but not limited to social networks, the web, finance, and natural science, has prompted the development of graph processing technologies, especially graph databases [22]. A number of query languages have emerged for graph database systems, including SPARQL for RDF graphs and Cypher, GQL, and SQL/PGQ for property graphs. The vast majority of graph query languages are based on graph pattern matching, in which a notable common component is conjunctive regular path queries (CRPQs) [4]. Intuitively, a CRPQ combines the features of subgraph matching, which matches a query subgraph to the data graph, and regular path queries (RPQs), which search for potentially variable-length paths whose label sequences conform to a given regular expression.

Regardless of their pivotal role in query languages and important use cases in application scenarios [4], there is currently no unified framework for optimizing CRPQs. A unified CRPQ optimization framework should consist of three core components:

- A **unified plan space** that provides a consistent set of operators applicable to both subgraph matching and RPQs;
- An **algebraic-rewriting-based plan enumeration** scheme utilizing equivalence-preserving rewriting rules, which enables a systematic and exhaustive search of the plan space to identify the globally optimal query plan;
- A **cost-based plan selection** algorithm that replaces fixed rules with adaptive cost estimations tailored to the specific characteristics of the CRPQ and the underlying data graph.

Such a comprehensive approach remains absent from the literature. Existing research and systems either optime subgraph matching [15, 27] and RPQs [21, 25] in isolation or rely on heuristic optimization techniques [2, 14], as summarized in Tab. 1.

Table 1: Comparison of our proposed framework against existing CRPQ optimization methods across key dimensions.

	Unified plan space	Algebraic-rewriting-based plan enumeration	Cost-based plan selection
Join-Ordering [2]	✗	✗	✓
MWJ-CRPQ [14]	✓	✗	✓
Neo4j [18]	✗	✗	✓
MillenniumDB [24]	✗	✗	✓
Ours	✓	✓	✓

In this paper, we propose a novel algebraic framework for executing and optimization CRPQs. Algebras for CRPQs exist [5, 7], but they represent CRPQs as the conjunction of RPQs, which does not fully exploit the relationship between the conjunction and the intra-RPQ operations, thus missing enhanced query evaluation and optimization opportunities.

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828624

The primary technical difficulty in divising a unified algebraic framework lies in the disparate plan representations for subgraph matching and RPQs. Specifically, subgraph matching plans are typically trees where the internal nodes denote join operations and the leaf nodes denote edge labels of the data graph [17]; while the plan representations of RPQs are based on either finite automata or extended relational algebra [25]. Such a gap has caused existing methods to separately plan and evaluate the subgraph matching and RPQ components in a CRPQ and finally fuse them according to a heuristic order, which is likely sub-optimal.

Despite the representation gap, we observe that there is an underlying similarity in how subgraph matching and RPQs are evaluated. Specifically, the concatenation in RPQs can be processed as continuous traversals on the data graph, which are analogous to left-deep joins. We abstract this basic operation as the *TI operator* (Def. 8), a core construct of CRPQs. Another core construct is the Kleene closure, the key feature of RPQs, which we abstract as the *KC operator* (Def. 9). In addition, we propose the *SJ operator* (Def. 7) to represent bushy joins during subgraph matching and the *UNION operator* (Def. 10) for handling RPQ alternation. Furthermore, we propose a novel hypergraph query model that enables composability of our query operators, generalizing existing graph algebras. This model addresses the long-standing challenge of lacking composability in existing ISO/IEC standardized graph query languages, such as GQL and SQL/PGQ [12, 13]. Based on our proposed hypergraph query model and operators, we devise a provably complete and composable algebra for CRPQs. The proposed algebraic operators have a direct mapping to their physical counterparts, which guarantees that composability is leveraged at the level of query optimization. We enumerate equivalent query plans based on algebraic transformation rules and devise a cost-based plan selection approach to efficiently select near-optimal plans from the rich plan space.

To verify our framework's effectiveness, we implement it based on open-source graph database systems, MillenniumDB [24] and Neo4j. On CRPQs extracted from WDBench [3] and LDBC SNB [8], our framework significantly outperforms the baseline systems.

In summary, we make the following contributions in this paper:

- We propose a unified algebraic optimization framework for conjunctive regular path queries (CRPQs) that jointly treats subgraph matching and regular path query (RPQ) components within a single, composable algebra, which bridges the representation gap between traditional tree-shaped subgraph matching plans and automata- or extended-relational-based RPQ plans.
- We devise a set of algebraic transformation rules to enumerate a rich space of equivalent CRPQ plans and design a cost-based optimization scheme that selects near-optimal plans over this unified space.
- We provide a full-fledged physical implementation of the algebra and optimizer, including concrete execution strategies for the proposed operators and cost and cardinality estimation methods.
- We integrate the framework into MillenniumDB and Neo4j and empirically demonstrate substantial performance gains on CRPQ workloads extracted from WDBench and LDBC SNB.

Roadmap. We introduce the preliminaries of CRPQs in Sec. 2. In Sec. 3, we present our proposed algebra for CRPQs, one of its feasible physical implementations, and how the algebraic operators combine to form query plans. We discuss the transformation rules that serve equivalent plan enumeration and a plan selection scheme based on cost and cardinality estimation in Sec. 4. Sec. 5 reports the experimental findings, Sec. 6 surveys the related work, and Sec. 7 concludes with our major findings and future directions.

2 PRELIMINARIES

To properly define conjunctive regular path queries (CRPQs), we first define their basic components, regular path queries (RPQs). Without specific clarification, we assume the queries discussed are posed to the directed edge-labeled graph $G = (V, E, \Phi, \lambda)$, where V is the vertex set, E is the edge set, Φ is the edge label set, and $\lambda : E \to \Phi$ is the assignment of the labels to each edge. We denote the set of query output variables as $\mathcal{V}$.

Definition 1 (RPQ)**.** An RPQ $R(x, y)$ has the following form:

$$R(x, y) \leftarrow \epsilon \mid a \mid a- \mid R_1/R_2 \mid R_1|R_2 \mid R? \mid R^* \mid R^+ \tag{1}$$

where $\langle x, y\rangle \in (\mathcal{V}\times\mathcal{V})\cup(\mathcal{V}\times V)\cup(V\times\mathcal{V})$, i.e., x and y are the data vertices (constants) or output variables corresponding to the source and target vertices of any matched path, which we omit when the context is clear. ϵ matches the empty path, i.e., $\{\langle v, v\rangle \mid v \in V\}$. a and $a-$ represent a single edge label and its inverse; we use a dedicated symbol $\diamond \notin \Phi$ to represent the wildcard, which matches all edge labels. $/, |, ?, *, +$ are the concatenation, alternation, zero-or-one, Kleene star, and Kleene plus operators, respectively. $/$ and $|$ are binary operators, and $?, *$ and $+$ are unary. $?, *$ and $+$ has the same level of precedence, followed by $/$ and lastly $|$.

Definition 2 (RPQ's result)**.** An RPQ $R(x, y)$'s result on the graph $G = (V, E, \Phi, l)$, where both x and y are variables, i.e., $x, y \in \mathcal{V}$, is a set of vertex pairs $[\![R(x, y)]\!]_G \subseteq V \times V$ defined as follows:

$$[\![a(x, y)]\!]_G = \{\langle u, v\rangle | \langle u, v\rangle \in E, l(\langle u, v\rangle) = a\} \text{ (Base case)} \tag{2}$$

$$[\![a\text{-}(x, y)]\!]_G = \{\langle v, u\rangle | \langle u, v\rangle \in E, l(\langle u, v\rangle) = a\} \text{ (Base case, inverse)} \tag{3}$$

$$[\![\diamond(x, y)]\!]_G = \{\langle u, v\rangle | \langle u, v\rangle \in E\} \text{ (Base case, wildcard)} \tag{4}$$

$$[\![(R_1/R_2)(x, y)]\!]_G = [\![R_1(x, z)]\!]_G \bowtie [\![R_2(z, y)]\!]_G \text{ (Concatenation)} \tag{5}$$

$$[\![(R_1|R_2)(x, y)]\!]_G = [\![R_1(x, y)]\!]_G \cup [\![R_2(x, y)]\!]_G \text{ (Alternation)} \tag{6}$$

$$[\![R?(x, y)]\!]_G = \{\langle v, v\rangle | v \in V\} \cup [\![R(x, y)]\!]_G \text{ (Zero-or-one)} \tag{7}$$

$$[\![R^+(x, y)]\!]_G = \cup_{i=1}^{\infty} [\![R^i(x, y)]\!]_G \text{ (Kleene plus)} \tag{8}$$

$$[\![R^*(x, y)]\!]_G = \{\langle v, v\rangle | v \in V\} \cup [\![R^+(x, y)]\!]_G \text{ (Kleene star)} \tag{9}$$

where R^i in Eq. 8 is the shorthand for concatenating R with itself for i times (e.g., $R^2 = R/R$).

When one of x and y is a constant $v \in V$, the RPQ's result is defined as follows, where σ denotes selection:

$$[\![R(v, y)]\!]_G = \pi_2(\sigma_{x=v}[\![R(x, y)]\!]_G) = \{\langle u\rangle \mid \langle v, u\rangle \in [\![R(x, y)]\!]_G\}$$

$$[\![R(x, v)]\!]_G = \pi_1(\sigma_{y=v}[\![R(x, y)]\!]_G) = \{\langle u\rangle \mid \langle u, v\rangle \in [\![R(x, y)]\!]_G\}$$

Definition 3 (CRPQ)**.** A CRPQ has the following form:

$$Q(\bar{x}) \leftarrow \bigwedge_{i=1}^{k} R_i(y_i, z_i), \text{ where } \bar{x} = \pi_{\{i_1,\cdots,i_{|\bar{x}|}\}} \hat{\oplus}_{j=1}^{k} (t_{j_1} \hat{\oplus} t_{j_2})$$

$$t_{j_1} = \begin{cases} \langle y_j\rangle, \text{if } y_j \in \mathcal{V} \\ \langle\rangle, \text{otherwise} \end{cases}, t_{j_2} = \begin{cases} \langle z_j\rangle, \text{if } z_j \in \mathcal{V} \\ \langle\rangle, \text{otherwise} \end{cases}, \tag{10}$$

where each $R_i(y_i, z_i)$ is an RPQ and $\bar{x}$ is the CRPQ's tuple of output variables. $\pi_{\{i_1,\cdots,i_{|\bar{x}|}\}}(\cdot)$ signifies projecting the elements indexed $\{i_1, \cdots, i_{|\bar{x}|}\}$ from the target tuple, and $\hat{\oplus}$ is the shorthand for tuple

concatenation with element deduplication, e.g., $\langle x_1, x_2\rangle \hat{\oplus} \langle x_2, x_3\rangle \hat{\oplus} \langle x_4\rangle = \langle x_1, x_2, x_3, x_4\rangle$.

CRPQs can be viewed as a generalization of subgraph matching, which specifies a query subgraph pattern to match onto the data graph. Any subgraph matching query can be expressed by Eq. 10 by fixing each R_i to be the label of each edge in the subgraph pattern. The fundamental difference between CRPQs and subgraph matching queries is that CRPQs allow edges in the subgraph pattern to be labeled by not only single edge labels, but also general RPQs.

Definition 4 (CRPQ's result)**.** A CRPQ $Q(\bar{x})$'s result on the graph $G = (V, E, \Phi, l)$ is denoted as $[\![Q(\bar{x})]\!]_G$ and defined as follows:

$$[\![Q(\bar{x})]\!]_G = \pi_{\bar{x}}(\bowtie_{i=1}^{k} [\![R_i(y_i, z_i)]\!]_G) \tag{11}$$

where $\bowtie$ denotes the natural join of the results of each RPQ on their common variables.

The above CRPQ definition aligns with the core query construct in widely used graph query languages, such as SPARQL, Cypher, and GQL.

Example 2.1. In Fig. 1, we give a running example of a CRPQ, which can be expressed according to Eq. 3 as

$$Q(v_0, v_1, v_2, v_3) \leftarrow a(v_1, v_2) \wedge a(v_2, v_3) \wedge b^+(v_1, v_0) \wedge b^+(v_2, v_0) \wedge b^+(v_3, v_0),$$

assuming all the query variables need to be returned. When this CRPQ is posed to the data graph in Fig. 2, it involves the subgraph marked with hatched vertices and bold edges, and produces the results in Fig. 2a. u_4 and u_5 do not appear in the results, since they are internal vertices in the paths mapped by RPQs.

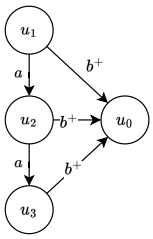


Figure 1: Example CRPQ.

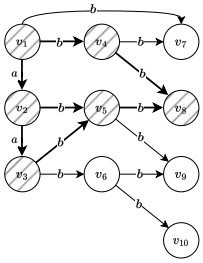
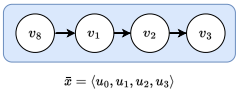


Figure 2: Example data graph.

u_0	u_1	u_2	u_3
v_8	v_1	v_2	v_3

(a) Table form (Def. 4).



(b) Hypergraph form (Def. 5).

Figure 3: Example CRPQ result.

3 UNIFIED CRPQ PLAN REPRESENTATION

In this section, we propose a unified CRPQ query plan representation that expands upon prior plan spaces. We first introduce a novel hypergraph-based query model as the basis of our logical operators (Sec. 3.1). Then, we present the logical query operators and their closedness and composability properties in Sec. 3.2, alongside an extensible set of physical implementation operators (3.3). Finally, Sec. 3.4 demonstrates how these operators combine to generate novel, efficient query plans unavailable in previous plan spaces.

3.1 Query Model: Results as Hypergraphs

We propose the following novel query model that represents a CRPQ's result by a hypergraph. We adopt the sequence hypergraph model, where each hyperedge consists of an ordered tuple of vertices [6].

Definition 5 (Result hypergraph of a CRPQ)**.** A CRPQ $Q(\bar{x})$'s result on the graph G can be alternatively defined as a hypergraph $\mathbf{G}^{\bar{x}}_{l_Q,G} = \langle \mathbf{V}^{\bar{x}}_{l_Q}, \mathbf{E}^{\bar{x}}_{l_Q}\rangle$, where $\mathbf{E}^{\bar{x}}_{l_Q} = \{\langle u_{i_1}, \cdots, u_{i_{|\bar{x}|}}\rangle_{l_Q} \mid \langle u_{i_1}, \cdots, u_{i_{|\bar{x}|}}\rangle \in [\![Q(\bar{x})]\!]_G\}$ and $\mathbf{V}^{\bar{x}}_{l_Q} = \{u \mid \exists e \in \mathbf{E}^{\bar{x}}_{l_Q}, \text{s.t. } \exists x_j \in \bar{x}, \langle u\rangle = \pi_{\{x_j\}}(e)\}$.

The result hypergraph $\mathbf{G}^{\bar{x}}_{l_Q,G}$ is $|\bar{x}|$-uniform, i.e., all of its hyperedges have $|\bar{x}|$ endpoints, each of which denoting a projected query vertex. All hyperedges in a result hypergraph $\mathbf{G}^{\bar{x}}_{l_Q,G}$ have the label l_Q, which is used to mark the type of the subquery that produces it as necessary for composability as discussed in Sec. 3.2.4. The set of all hypergraph labels, denoted as Σ, is a superset of the data graph's original and inverse edge label set $\overleftrightarrow{\Phi}$.

We use the hypergraph result form throughout the paper. Without further specification, the notation $[\![Q]\!]_G$ refers to Q's result hypergraph $\mathbf{G}^{\bar{x}}_{l_Q,G}$ hereafter.

Example 3.1. Fig. 2b shows the result hypergraph of the CRPQ in Fig. 1, which consists of a single hyperedge $\langle u_8, u_1, u_2, u_3\rangle$, preserving the mapping from the output variables $\langle v_0, v_1, v_2, v_3\rangle$. The arrows between the vertices in a hyperedge denote their ordered relation. The hyperedge label is omitted without ambiguity. Note that the result hypergraph differs from the data subgraph involved with the result shown in Fig. 2.

Our hypergraph query model natively supports query composability, allowing outputs to serve directly as inputs for subsequent queries. Composability requires matching input and output data types, which is a long-standing challenge in graph querying [10]. Existing languages like GQL and SQL/PGQ output tables rather than queryable graphs, which either breaks the compositional chain or necessitates intermediate graph views. In contrast, our algebra seamlessly chains graph inputs and outputs without requiring intermediate representations. The initial data graph is ported into our algebra via the SCAN operator, which extracts all edges with a specific label to construct a hypergraph compliant with Def. 5. We believe our proposal goes beyond the path-based composability of path-oriented algebras [5] and suggests a promising design principle for future versions of standardized graph query languages.

3.2 Logical Query Operators

3.2.1 Motivation. In this paper, we design an algebra for CRPQ evaluation. Our design is primarily motivated by the observation that *there is an underlying similarity between how RPQs and their conjunction are evaluated.* Specifically, each RPQ evaluation step for concatenation (/) consists of extending the currently matched path by traversing from it an edge or another path, while the conjunction of multiple RPQs can be evaluated in a similar manner: extending–or filtering, when both the source and target vertices of the traversed edges and paths are already matched–the currently matched path or subgraph by traversing one or more edge or path, when there are multiple edges or paths traversed, intersecting on the target vertex. This evaluation method is fundamentally different from evaluating

Table 2: Notations used across the paper.

Notation	Description
$G = (V, E, \Phi, \lambda)$	A data graph
$\mathbf{G}_l^{\bar{x}} = \langle \mathbf{V}_{\mathbf{G}_l^{\bar{x}}}, \mathbf{E}_{\mathbf{G}_l^{\bar{x}}} \rangle$	A directed hypergraph whose edges are labeled with l; $V_{\mathbf{G}_l^{\bar{x}}}$ and $E_{\mathbf{G}_l^{\bar{x}}}$ are its sets of vertices and edges, respectively
$\hat{\oplus}$	The shorthand for tuple concatenation with element deduplication, e.g., $\langle x_1, x_2 \rangle \hat{\oplus} \langle x_2, x_3 \rangle = \langle x_1, x_2, x_3 \rangle$
$\bar{x}$	The output variables of a CRPQ, to which the endpoints of the hyperedges in a result hypergraph are mapped in order
Σ	The set of hyperedge labels
Σ_p	The set of hyperedge labels for RPQ results
$\overleftrightarrow{\Phi}$	The set of original and inverse edge labels in the data graph, i.e., $\overleftrightarrow{\Phi} = \Phi \cup \{a- \mid a \in \Phi\}$

the conjunction of RPQs by combining matched subgraphs, which have an arbitrary number of common query vertices, in the style of relational joins. These two logical operators call for distinct physical implementation techniques. That is why we define two logical operators in the algebra to properly distinguish between them: the **Traverse-Intersect (TI, Def. 8)** operator to capture the traversal-style evaluation, and the **Subgraph Join (SJ, Def. 7)** operator for the relational-join-style evaluation.

Besides concatenation, RPQs also comprise Kleene closures (* and +) and alternation (|). We define the **Kleene closure (KC, Def. 9)** operator for * or +. The **UNION (Def. 10)** operator is defined for |. In addition, **INV** facilitates generating plans for query edges and paths in both directions for TI, and **SCAN**, which retrieves all edges with a given label from the data graph, serves as the algebra's entry point. Fig. 4 shows the correspondence between the query constructs in CRPQs and their respective logical operators.

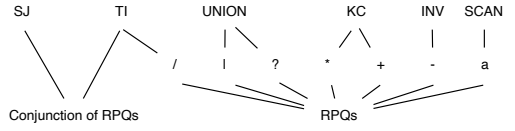


Figure 4: Correspondence between the query constructs in CRPQs and the logical operators.

In the following, we introduce the formal definition of each logical operator and use examples to illustrate their semantics. Lastly, we discuss the proposed algebra's closedness and composability.

3.2.2 Definition of Operators. The notation of the logical operators defined in this paper is $\mathrm{Op}^{\bar{x},d}$, where $\bar{x}$ is the tuple of output variables and d is the hyperedge label of the result hypergraph. Due to limited space, we present only the edge sets of the result hypergraphs in the following logical operator definitions, since any result hypergraph's vertex set can be derived from its edge set by Def. 5.

Definition 6 (SCAN)**.** The **SCAN operator**, denoted $SCAN^{\langle x,y\rangle,a}(G)$ or $SCAN^{\langle x,y\rangle,a-}(G)$, where $G = (V, E, \Phi, \lambda)$ is the data graph and $a \in \Phi$, is used to retrieve the data subgraph consisting of all edges labeled a and is the entry point of our proposed algebra. Specially, for the wildcard label $\diamond$, all edges in the data graph are retrieved. Formally:

$$\mathbf{E}_{SCAN^{\langle x,y\rangle,a}(G)} = \{\langle v, v'\rangle_a \mid \langle v, v'\rangle \in E, \lambda(\langle v, v'\rangle) = a\}, \quad (12)$$

$$\mathbf{E}_{SCAN^{\langle x,y\rangle,a-}(G)} = \{\langle v', v\rangle_a \mid \langle v, v'\rangle \in E, \lambda(\langle v, v'\rangle) = a\}. \quad (13)$$

$$\mathbf{E}_{SCAN^{\langle x,y\rangle,\diamond}(G)} = \{\langle v, v'\rangle_\diamond \mid \langle v, v'\rangle \in E\}. \quad (14)$$

Definition 7 (Subgraph Join (SJ))**.** The **subgraph join (SJ) operator** is defined as $SJ^{\bar{x},d}(\mathbf{G}_{l_1}^{\overline{x_1}}, \mathbf{G}_{l_2}^{\overline{x_2}})$, where $\mathbf{G}_{l_1}^{\overline{x_1}}$ (or $\mathbf{G}_{l_2}^{\overline{x_2}}$) is a directed hypergraph whose edges are labeled with l_1 (or l_2) and $l_1, l_2 \in \Sigma - \overleftrightarrow{\Phi} - \Sigma_p$, i.e., both operands denote the results of query subgraphs; $\bar{x} \subseteq \bar{x_1} \hat{\oplus} \overline{x_2}$ indicates the endpoints of the resulting hypergraph, i.e., the query vertices to project; and $d \in \Sigma - \overleftrightarrow{\Phi} - \Sigma_p$ represent the label of the resulting hyperedges, marking the results as those of a query subgraph. Formally:

$$\begin{aligned}
&\mathbf{E}_{SJ^{\bar{x},d}(\mathbf{G}_{l_1}^{\overline{x_1}}, \mathbf{G}_{l_2}^{\overline{x_2}})} = \{\langle v_{i_1}, \cdots, v_{i_{|\bar{x}|}}\rangle_d \mid \exists \langle v_{j_1}, \cdots, v_{j_{|\overline{x_1}|}}\rangle_{l_1} \in \mathbf{E}_{\mathbf{G}_{l_1}^{\overline{x_1}}}, \\
&\langle v_{k_1}, \cdots, v_{k_{|\overline{x_2}|}}\rangle_{l_2} \in \mathbf{E}_{\mathbf{G}_{l_2}^{\overline{x_2}}}, \text{s.t. } \langle v_{j_1}, \cdots, v_{j_{|\overline{x_1}|}}\rangle = \pi_{\overline{x_1}}(\langle v_{i_1}, \cdots, v_{i_{|\bar{x}|}}\rangle), \\
&\langle v_{k_1}, \cdots, v_{k_{|\overline{x_2}|}}\rangle = \pi_{\overline{x_2}}(\langle v_{i_1}, \cdots, v_{i_{|\bar{x}|}}\rangle)\}, \qquad (15) \\
&\text{where } d \in \Sigma - \overleftrightarrow{\Phi} - \Sigma_p,\ \bar{x} \subseteq \overline{x_1} \hat{\oplus} \overline{x_2}.
\end{aligned}$$

Definition 8 (Traverse-Intersect (TI))**.** The **traverse-intersect (TI) operator** is defined as $TI^{\bar{x},d}(\mathbf{G}_{l_L}^{\overline{x_L}}, \mathbf{G}_{l_1}^{\overline{x_1}}, \cdots, \mathbf{G}_{l_n}^{\overline{x_n}})$, where all the operand hypergraphs except the leftmost one, $\mathbf{G}_{l_L}^{\overline{x_L}}$, denote single edges or paths. Intuitively, all these edges or paths are *traversed*, and the data vertices they respectively reach, if not originally constants in the query, are then *intersected* to get the results. Only when there are exactly two operands that are edges or paths strung together by a common variable is the resulting d in Σ_p, since this signifies traversing from a single edge or a path to another one, forming a path; in all other cases, $d \in \Sigma - \overleftrightarrow{\Phi} - \Sigma_p$. Formally:

$$\begin{aligned}
&\mathbf{E}_{TI^{\bar{x},d}(\mathbf{G}_{l_L}^{\overline{x_L}}, \mathbf{G}_{l_1}^{\overline{x_1}}, \cdots, \mathbf{G}_{l_n}^{\overline{x_n}})} \\
&= \{\langle v_{i_1}, \cdots, v_{i_{|\bar{x}|}}\rangle_d \mid \exists \langle v_{j_1}, \cdots, v_{j_{|\overline{x_L}|}}\rangle_{l_L} \in \mathbf{E}_{\mathbf{G}_{l_L}^{\overline{x_L}}}, \\
&\bigwedge_{m=2}^{n} \langle v_{k_1}, \cdots, v_{k_{|\overline{x_m}|}}\rangle_{l_m} \in \mathbf{E}_{\mathbf{G}_{l_m}^{\overline{x_m}}}, \\
&\text{s.t. } \langle v_{j_1}, \cdots, v_{j_{|\overline{x_1}|}}\rangle = \pi_{\overline{x_1}}(\langle v_{i_1}, \cdots, v_{i_{|\bar{x}|}}\rangle), \\
&\bigwedge_{m=2}^{n} \langle v_{k_1}, \cdots, v_{k_{|\overline{x_m}|}}\rangle = \pi_{\overline{x_m}}(\langle v_{i_1}, \cdots, v_{i_{|\bar{x}|}}\rangle)\}, \qquad (16) \\
&\text{with the following constraints:} \\
&\textbf{Query vertex constraint: } \overline{x_i} = \langle x_i\rangle \vee \overline{x_i} = \langle x_i, y\rangle, \\
&\bigcup_{|\overline{x_i}|=2} \{x_i\} \subseteq \overline{x_L}, x_i, y \in \mathcal{V}, \bar{x} \subseteq \hat{\oplus}_{m=1}^{n} \overline{x_m}; \qquad (17) \\
&\textbf{Hypergraph label constraint: } l_i \in \overleftrightarrow{\Phi} \cup \Sigma_p,\ \forall i \in \{1, \cdots, n\}, \\
&d \in \begin{cases} \Sigma_p, \text{ if } n = 1, \{l_1, l_2\} \subseteq \overleftrightarrow{\Phi} \cup \Sigma_p, \pi_{|\overline{x_L}|}(\overline{x_L}) = x_1 \\ \Sigma - \overleftrightarrow{\Phi} - \Sigma_p, \text{ otherwise.} \end{cases} \qquad (18)
\end{aligned}$$

Definition 9 (Kleene Closure (KC))**.** The **Kleene Closure (KC) operator** is defined as $KC^{\langle x,y\rangle,d}(\mathbf{G}_l^{\langle x,y\rangle})$, which corresponds to the RPQ l^+, where $x, y \in \mathcal{V}$ and $d \in \Sigma_p$. The annotation below KC can

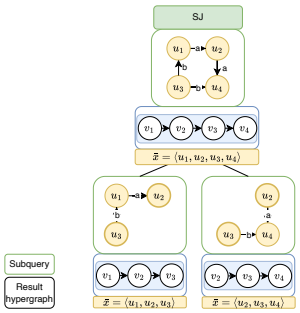


Figure 5: Example SJ operator.

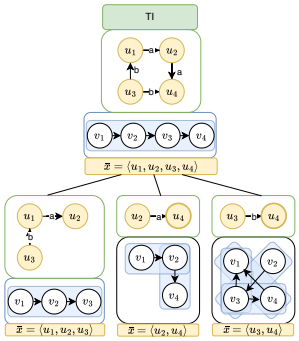


Figure 6: Example TI operator.

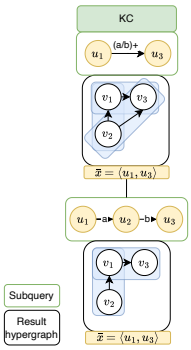


Figure 7: Example KC operator.

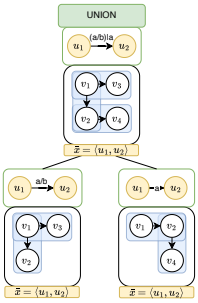


Figure 8: Example Union operator.

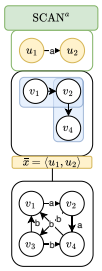


Figure 9: Example SCAN operator.

be + or ∗. Formally:

$$\mathbf{E}_{KC^{\langle x,y\rangle,d}_{+}(\mathbf{G}_l^{\langle x,y\rangle})} = \bigcup_{i\in\mathbb{N}} U_i, \text{ where } d \in \Sigma_p \tag{19}$$

$$U_1 = \mathbf{E}_{\mathbf{G}_l^{\langle x,y\rangle}} \tag{20}$$

$$U_{i+1} = \mathbf{E}_{TI^{\langle x,y\rangle,d}(\rho_{x'/y}(U_i),\rho_{x'/x}(U_1))} \tag{21}$$

$$\mathbf{E}_{KC^{\langle x,y\rangle,d}_{*}(\mathbf{G}_l^{\langle x,y\rangle})} = \mathbf{E}_{KC^{\langle x,y\rangle,d}_{+}(\mathbf{G}_l^{\langle x,y\rangle})} \cup \{\langle v,v\rangle_d \mid v \in V_{KC^{\langle x,y\rangle,d}_{+}(\mathbf{G}_l^{\langle x,y\rangle})}\}, \tag{22}$$

where $\rho_{x'/x}$ is the standard rename operation that renames the variable x to x' in relational algebra, allowing TI to operate on U_i and U_1 in the iterative step.

Definition 10 (UNION). The **union operator** $UNION^{\bar{x},d}$ merges two result hypergraphs $\mathbf{G}_{l_1}^{\overline{x_1}}, \mathbf{G}_{l_2}^{\overline{x_2}}$ and assigns the new label d to the hyperedges in the merged hypergraph. Since UNION is only used for evaluating alternation in RPQs in the scope of CRPQs, it requires that $\overline{x_1} = \overline{x_2}$, $l_1 \in \Sigma_p \cup \overleftrightarrow{\Phi}$, and $l_2 \in \Sigma_p \cup \overleftrightarrow{\Phi}$ and produces $d \in \Sigma_p$. Formally:

$$\mathbf{E}_{UNION^{\bar{x},d}(\mathbf{G}_{l_1}^{\overline{x_1}},\mathbf{G}_{l_2}^{\overline{x_2}})} = \{\langle v_{i_1},\cdots,v_{i_{|\bar{x}|}}\rangle_d \mid \langle v_{i_1},\cdots,v_{i_{|\bar{x}|}}\rangle_{l_1} \in \mathbf{E}_{\mathbf{G}_{l_1}^{\overline{x_1}}}\} \cup \{\langle v_{i_1},\cdots,v_{i_{|\bar{x}|}}\rangle_d \mid \langle v_{i_1},\cdots,v_{i_{|\bar{x}|}}\rangle_{l_2} \in \mathbf{E}_{\mathbf{G}_{l_2}^{\overline{x_2}}}\}, \tag{23}$$

$$\text{where } d \in \Sigma_p,\ \bar{x} = \overline{x_1} = \overline{x_2}.$$

Definition 11 (Inverse (INV)). The **inverse (INV) operator** is defined as $INV^{\langle x,y\rangle,d}(\mathbf{G}_l^{\langle x,y\rangle})$. INV is used for pushing down constants from KC and preparing non-leftmost operands that satisfy the constraints for TI, so it requires $l \in \Sigma_p \cup \overleftrightarrow{\Phi}$. Formally:

$$\mathbf{E}_{INV^{\langle x,y\rangle,d}(\mathbf{G}_l^{\langle x,y\rangle})} = \{\langle v,v'\rangle_d \mid \langle v',v\rangle_l \in E_{\mathbf{G}_l^{\langle x,y\rangle}}\}. \tag{24}$$

The remaining CRPQ operation, zero-or-one (?), does not require a dedicated logical operator, since it directly translates into $UNION^{\bar{x},d}(\mathbf{G}_l^{\bar{x}}, \langle \mathbf{V}_{\mathbf{G}_l^{\bar{x}}}, \{\langle v,v\rangle_l \mid v \in \mathbf{V}_{\mathbf{G}_l^{\bar{x}}}\}\rangle)$ on the hypergraph $\mathbf{G}_l^{\bar{x}}$.

To express the constraints of SJ and TI on the operand hypergraphs' types, it is necessary to mark whether a hypergraph is the result of a query subgraph or path by using different sets of hypergraph edge labels, because a hypergraph's cardinality is not indicative of its subquery type. For example, it is possible for a query subgraph to project only two query vertices.

Examples. Fig. 5 to 9 are examples of each logical operator. The logical operators are shown as green nodes attached to their query graphs, hypergraph results, and output variables from top to bottom. Their operands are displayed as these nodes' children.

3.2.3 Treatment of Constants. According to Def. 1 and 3, the vertices in a CRPQ query graph, i.e., the sources or targets of RPQs, can be variables or constants. Therefore, the logical operators that can produce RPQ results, which are SCAN, TI, KC, and UNION, need to handle constants. To represent the constant endpoints of RPQs, we introduce a variable-constant mapping in the operator subscript, $\mathrm{Op}^{\bar{x},d}_{x'\mapsto a}$, where $x' \notin \bar{x}$. The results of an operator with such a mapping are defined as follows:

$$\mathbf{E}_{\mathrm{Op}^{\bar{x},d}_{x'\mapsto a}(\cdot)} = \{\pi_{\bar{x}}(e) \mid e \in \mathbf{E}_{\mathrm{Op}^{\bar{x}\oplus\langle x'\rangle,d}(\cdot)}, \pi_{x'}(e) = a\} \tag{25}$$

We will further discuss how the constants can be pushed down to the operators' operands in the transformation rules (Sec. 4.1.1).

3.2.4 Closedness & Composability. Closedness refers to the property that the output of each operator falls in the input domain of the algebra. As explained in Sec. 3.1, our algebra's input domain is the set of $|\bar{x}|$-uniform sequence hypergraphs in which all hyperedges have the same label. We can prove that each of our proposed logical operators has the closedness property. Due to limited space, we present the theorem for the SJ operator's closedness and its proof as follows, which can be extended to the other operators.

Theorem 1. The SJ opeator has the closedness property.

Proof. For convenience, we denote $SJ^{\bar{x},d}(\mathbf{G}_{l_1}^{\overline{x_1}}, \mathbf{G}_{l_2}^{\overline{x_2}})$ as $\mathbf{G}_{SJ}$ in this proof for any given $\mathbf{G}_{l_1}^{\overline{x_1}}$ and $\mathbf{G}_{l_2}^{\overline{x_2}}$. According to Eq. 15 in Def. 7, each hyperedge in $\mathbf{G}_{SJ}$ is of the form $\langle u_{i_1},\cdots,u_{i_{|\bar{x}|}}\rangle_d$, which means that $\mathbf{G}_{SJ}$ is a $|\bar{x}|$-uniform sequence hypergraph in which all hyperedges have the label d. Therefore, any SJ is in the input domain of the SJ operator, that is, SJ has the closedness property. □

Composability allows algebraic operators to be chained, i.e., the output of one serves as the input for another, with the final result remaining in the same domain. Fig. 11 shows the composability relations between our logical operators. The hyperedge labels serve as intermediaries for composable operators. An operator is composable with another if and only if there is a path between them. For example, KC is directly composable with TI, since a KC's result

can serve as one of the non-leftmost operands for TI, but KC is only composable with SJ via TI.

Fig. 10 offers a concrete example of composable algebraic operators and their correspondence to language-level constructs. The right half shows equivalent CRPQs in SPARQL, Cypher, and ISO/GQL. The left half shows a possible algebraic representation of the query graphically and formally. The query models of all three query languages return tables from querying graphs, while our proposed hypergraph result model ensures that the intermediate result of each processing step is a hypergraph, which generalizes the graph model of the input data, offering closedness and composability. Closedness and composability are highly desirable properties for an algebra, since they help derive transformation rules for query optimization, as detailed in Sec. 4.1.1.

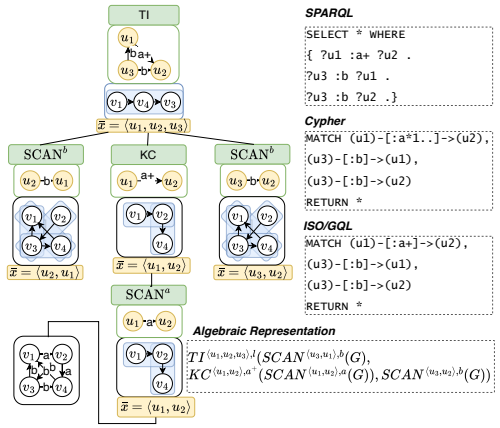


Figure 10: Example of composable algebraic operators and their correspondence to language-level constructs.

3.3 Physical Query Operators

A physical query operator is a physical implementation of a logical query operator. In this paper, we focus on the physical operators introduced below to enable a concrete physical implementation of our framework, without attempting to be exhaustive. The correspondence relations between the logical and physical operators are shown in Fig. 3. INV does not have a corresponding physical operator, since it is always pushed through all other logical operators until reaching SCAN, which handles inverse edge labels natively.

In addition to the working principles of each physical operator, we also discuss how the physical operators for SCAN and KC handle constants, as required by Sec. 3.2.3. The physical operators for TI and UNION do not need to handle constants, as their constants are always pushed down to the operands according to the transformation rules (Sec. 4.1.1).

3.3.1 B+-Tree-Based Scan (BPT-SCAN). We assume the data graph is indexed in two B+-trees: one indexed by (edge label, source, target) and the other by (edge label, target, source), as in our experimental baseline system, MillenniumDB. A SCAN, given an edge label or its inverse, uses a standard partial-key lookup on the first or second B+-tree, respectively.

Table 3: Logical & physical operators' correspondence.

Logical operator	Physical operator
SCAN	BPT-SCAN
SJ	HJ
TI	LFJ
KC	CAKC
UNION	MU

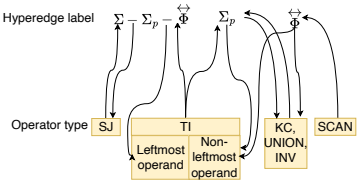


Figure 11: The composability relations between our logical operators.

BPT-SCAN's constant handling. When the source or target of a BPT-SCAN is a constant, it is incorporated into the key for lookup. Specifically, if the target is constant, it is treated as the source, so the second B+-tree is used for original edge labels, while the first is used for inverses.

3.3.2 Hash Join (HJ). HJ is a well-known physical join implementation, which we use to implement SJs. It takes two tables as input and outputs the result of their natural join. Its working principle involves iterating through each row of the outer table and using a hash table on the inner table to quickly find matching rows. In our context, the operands of the HJ are result hypergraphs, where each hyperedge corresponds to a table row.

3.3.3 Leapfrog Join (LFJ). LFJ [23] takes as input the two or more tables and outputs the result of their natural join. It leverages the sorted indexes on these result tables: by alternating across the indexes using leapfrog pointers, it efficiently avoids unnecessary data accesses. The LFJ requires that all the operand result tables have sorted indexes on the common columns. If such indexes are missing, they must be built before the LFJ's execution.

In this paper, we use LFJs for TIs. All operands of a TI operator except the leftmost represent edges or paths in the query graph. For each hyperedge in the leftmost operand's hypergraph, neighbor lists of its common query vertices with other operands are produced and sorted on demand for an LFJ. Non-leftmost operands are SCAN, TI, KC, or UNION. SCAN results come pre-sorted from a B+-tree, while others are computed and sorted lazily, with mapped constants of the common vertices from the leftmost operand pushed down to drive their execution.

After intersection, if the intersected query vertex is not in the TI's leftmost operand, the output of the LFJ is the set of matches of the intersected query vertex, each of which combines with the current hyperedge in the leftmost operand to form a hyperedge in this TI's results. Otherwise, we do a binary search over the output of the LFJ using the match of the intersected query vertex in the current hyperedge as key, and preserve this hyperedge in the result hypergraph if and only if the search returns true.

3.3.4 Cache-Aided Kleene Closure (CAKC). CAKC is a physical query operator for implementing the KC operator. For the RPQ R^* or R^+, CAKC caches the intermediate query results of R obtained during its child plan's initial evaluation to avoid recomputation in subsequent iterations.

In each subsequent iteration, newly produced targets from the previous iteration are pushed down as constant sources to the

cached R to be joined, and new hyperedges formed from prior sources and their respective current targets are added to the result hypergraph if not already present. The iterations continue until no new results are produced.

When the KC operator implemented by CAKC is $*$, its result set includes $\{\langle v, v\rangle \mid v \in V\}$. To avoid immediately materializing this large set during execution, the lazy evaluation mechanism used for handling ϵ's in Pang et al. [21] is applied. Specifically, an ϵ marker is attached to the corresponding result set to be processed by the parent operator instead of explicitly including the aforementioned set, which saves space and time.

CAKC's constant handling. When a KC operator has a constant as its source, it cannot be directly pushed down to its operand in a transformation rule like TI and UNION, or each iteration would operate on only a partial view of the data, leading to incomplete results. Thus, CAKC pushes down the constant source *only in the initial iteration*, constraining the sources of the final result and thus reducing the computation cost. However, CAKC is unable to conduct any pushdown for constant targets. Therefore, we apply an INV operator to the KC's operand to continue using the same strategy whenever possible:

$$\mathbf{E}_{KC^{\langle x\rangle,d}_{y\mapsto a}(\cdot)} = \{\pi_x(e) \mid e \in \mathbf{E}_{INV^{\langle x,y\rangle}(KC^{\langle x,y\rangle,d}(\cdot))}, \pi_y(e) = a\} \quad (26)$$

3.3.5 Merge Union (MU). MU is a physical query operator used to implement the alternation operation (|). It scans each hyperedge in the inputs and merges them to produce a final result hypergraph.

3.4 CRPQ Query Plans

In this section, we introduce how the aforementioned operators combine to form CRPQ query plans.

3.4.1 Logical Plans. A logical plan for a CRPQ is a tree made up of logical query operators and their operands, which dictates their evaluation order.

Definition 12 (CRPQ's Logical Plan)**.** A CRPQ's logical plan is structured as a tree, where the nodes represent logical query operators, whose child nodes are the operators that produce their operand result hypergraphs. The leaf nodes must be SCAN operators; we omit their operands, which is the data graph.

Example 3.2. Fig. 12 presents an example CRPQ logical plan corresponding to the CRPQ query graph in Fig. 1.

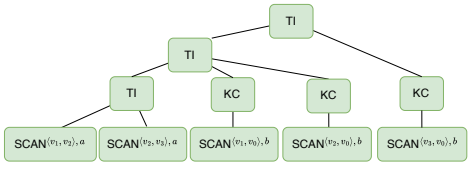


Figure 12: Example logical plan of the CRPQ in Fig. 1.

3.4.2 Expressivity of the Logical Plan Space. To prove that the logical plan space is sufficient to express any CRPQ, an algorithm can be provided to convert any given CRPQ into a canonical logical query plan in our plan space. Due to the page limit, we delay the pseudocode to the appendix[1] and provide a brief description here. For each RPQ in the input CRPQ, we call a subroutine to get its canonical subplan, then join the subplans one by one using TI. We ensure the sources of non-leftmost operands to be a common variable with the leftmost operand according to TI's definition (Def. 8) by applying an INV operator when necessary. The subroutine removes the highest-priority operation from the input RPQ, processes the remainder recursively, and combines the returned plans with the logical operator corresponding to the stripped operation.

3.4.3 Physical Plans. A CRPQ's physical plan is derived from its logical plan by selecting a specific physical query operator to implement each logical query operator. In the physical plan tree, each logical operator node is replaced with its corresponding physical operator. For example, with our proposed physical query operators (Sec. 3.3), Fig. 12's corresponding physical plan would be all the TIs replaced with LFJs, all the KCs replaced with CAKCs, and all the SCANs replaced with BPT-SCANs.

A physical plan can be evaluated with a post-order traversal, where each encountered physical operator node is evaluated with the result hypergraphs produced by its child nodes as input. The result hypergraph produced by the root node is the final result of the entire query.

3.4.4 Novel & Efficient Plans in the Proposed Plan Space. In the following, we illustrate by an example that our proposed plan space includes novel, efficient query plans.

Example 3.3. Fig. 1 shows an example CRPQ query graph, and Fig. 2 shows the data graph on which the CRPQ is to be evaluated. It is optimal to first match edges labeled a, as the data graph contains many b^+ results. Then, using a TI operator with two KC operands to evaluate the b^+ edges from u_1 and u_2 to u_0 narrows the intermediate results to a single match. A second TI can then efficiently combine the KC operand for the b^+ edge from u_3 to u_0, avoiding the costlier alternative of combining all three b^+ edges in one TI, which would require computing b^+ from v_3 and involve more data edges. This plan, as shown in Fig. 12, is also more efficient than those that use SJ, since any plan that uses SJ will repeatedly evaluate b^+ from at least one data vertex.

Previous approaches cannot express this plan, which is flexible in that it allows query edges that share a vertex, e.g., $\langle u_1, u_0\rangle$, $\langle u_2, u_0\rangle$, and $\langle u_3, u_0\rangle$, to be handled separately. It also allows multiple RPQs to participate in a single join, and imposes no constraints on the order in which edges from the query graph appear in the plan.

4 UNIFIED CRPQ PLANNING

In the previous section, we introduce a unified plan space for CRPQs and show that it includes efficient plans not expressible by previous works. Since a CRPQ may have multiple semantically equivalent plans in the plan space, it is necessary to find the near-optimal plan for evaluation with a plan selection scheme, which involves plan enumeration (Sec. 4.1) and cost estimation (Sec. 4.2).

4.1 Plan Enumeration & Selection

This section presents transformation rules for generating equivalent query plans based on the logical query operators' definitions

[1] https://github.com/SoftlySpoken/CRPQ-unified-planning/blob/crpq/full_paper.pdf

and introduces a unified algorithm (Alg. 2) that uses these rules to enumerate feasible plans and select the one with the lowest estimated cost for execution.

4.1.1 Transformation Rules. Transformation rules, when applied to a query plan, transform it into equivalent plans with varying efficiency. Alg. 2 invokes the transformation rules, which are derived from the definitions of the logical operators, to systematically explore the logical plan space and select efficient plans for execution. In the following, we present the transformation rules and how they apply to query planning.

TI's right-recursive reducibility. The TI operator is reducible right-recursively, which means that a nested TI can be added for the rightmost operand:

$$\begin{aligned} &TI^{\bar{x},d}(\mathrm{G}_{l_L}^{\overline{x_L}}, \mathrm{G}_{l_1}^{\overline{x_1}}, \cdots, \mathrm{G}_{l_n}^{\overline{x_n}}) \\ =&TI^{\bar{x},d}(TI^{\overline{x_L}\hat{\oplus}(\hat{\oplus}_{m=1}^{n-1}\overline{x_m}),d'}(\mathrm{G}_{l_L}^{\overline{x_L}}, \mathrm{G}_{l_1}^{\overline{x_1}}, \cdots, \mathrm{G}_{l_{n-1}}^{\overline{x_{n-1}}}), \mathrm{G}_{l_n}^{\overline{x_n}}). \end{aligned} \tag{27}$$

This rule is directly inferrable from TI's definition (Def. 8). Since all the TI operands except the leftmost one are equal in status and can be reordered according to TI's definition, any operand except for the leftmost one can be peeled off in this way. We can also apply this rule conversely to merge an operand of the outer TI to the inner TI as long as the TI query vertex and hypergraph label constraints (Eq. 17 and 18) are conformed. Alg. 2 applies this rule by enumerating the query edges and paths as operands of the TI operator (line 6).

SJ's associativity. The SJ operator is associative, which is inferrable from its definition:

$$\begin{aligned} &SJ^{\bar{x_1}\hat{\oplus}\bar{x_2}\hat{\oplus}\bar{x_3},d}(SJ^{\bar{x_1}\hat{\oplus}\bar{x_2},d_2}(\mathrm{G}_{l_1}^{\bar{x_1}}, \mathrm{G}_{l_2}^{\bar{x_2}}), \mathrm{G}_{l_3}^{\bar{x_3}}) \\ =&SJ^{\bar{x_1}\hat{\oplus}\bar{x_2}\hat{\oplus}\bar{x_3},d}(\mathrm{G}_{l_1}^{\bar{x_1}}, SJ^{\bar{x_2}\hat{\oplus}\bar{x_3},d_3}(\mathrm{G}_{l_2}^{\bar{x_2}}, \mathrm{G}_{l_3}^{\bar{x_3}})). \end{aligned} \tag{28}$$

This rule leads to enumerating different ways of joining two query subgraphs to obtain the results of the same query subgraph in Alg. 2 (line 15).

Constant pushdown. As mentioned in Sec. 3.2.3, constant sources or targets of RPQs whose lowest-precedence operator is TI or UNION can be pushed down to their operands. In this case, the TI operator must satisfy the constraints (Eq. 17 and 18). Alg. 2 applies these rules whenever possible, since they reduce the size of intermediate results and consequently lowers evaluation cost.

$$\begin{aligned} &TI_{x\mapsto v}^{\langle x,y\rangle,d}\left(\mathrm{Op}^{\langle x,z\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}^{\langle z,y\rangle,l_2}(\mathrm{G}_2)\right) \\ =&\,TI^{\langle y\rangle,d}\left(\mathrm{Op}_{x\mapsto v}^{\langle x,z\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}^{\langle z,y\rangle,l_2}(\mathrm{G}_2)\right). \end{aligned} \tag{29}$$

$$\begin{aligned} &TI_{y\mapsto v}^{\langle x,y\rangle,d}\left(\mathrm{Op}^{\langle x,z\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}^{\langle z,y\rangle,l_2}(\mathrm{G}_2)\right) \\ =&\,TI^{\langle x\rangle,d}\left(\mathrm{Op}^{\langle x,z\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}_{y\mapsto v}^{\langle z\rangle,l_2}(\mathrm{G}_2)\right). \end{aligned} \tag{30}$$

$$\begin{aligned} &UNION_{x\mapsto v}^{\langle x,y\rangle,d}\left(\mathrm{Op}^{\langle x,y\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}^{\langle x,y\rangle,l_2}(\mathrm{G}_2)\right) \\ =&\,UNION^{\langle y\rangle,d}\left(\mathrm{Op}_{x\mapsto v}^{\langle x,y\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}_{x\mapsto v}^{\langle x,y\rangle,l_2}(\mathrm{G}_2)\right). \end{aligned} \tag{31}$$

$$\begin{aligned} &UNION_{y\mapsto v}^{\langle x,y\rangle,d}\left(\mathrm{Op}^{\langle x,y\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}^{\langle x,y\rangle,l_2}(\mathrm{G}_2)\right) \\ =&\,UNION^{\langle x\rangle,d}\left(\mathrm{Op}_{y\mapsto v}^{\langle x,y\rangle,l_1}(\mathrm{G}_1), \mathrm{Op}_{y\mapsto v}^{\langle x,y\rangle,l_2}(\mathrm{G}_2)\right). \end{aligned} \tag{32}$$

INV pushdown. An INV operator that operates on TI, KC, or UNION can be pushed down to their operands, while on a SCAN, it equates to a label inverse:

$$INV^{\langle x,y\rangle,d}(SCAN^{\langle x,y\rangle,a}(G)) = SCAN^{\langle x,y\rangle,a^-}(G). \tag{33}$$

$$\begin{aligned} &INV^{\langle x,y\rangle,d}\left(TI^{\langle x,y\rangle,d'}(\mathrm{Op}^{\langle x,z\rangle,d_1}(\mathrm{G}_1), \mathrm{Op}^{\langle z,y\rangle,d_2}(\mathrm{G}_2))\right) \\ =&\,TI^{\langle x,y\rangle,d}\Big(\rho_{z/y}\circ\rho_{x/z}(INV^{\langle z,y\rangle,d_2'}(\mathrm{Op}^{\langle z,y\rangle,d_2}(\mathrm{G}_2))), \\ &\,\rho_{z/x}\circ\rho_{y/z}(INV^{\langle x,z\rangle,d_1'}(\mathrm{Op}^{\langle x,z\rangle,d_1}(\mathrm{G}_1)))\Big). \end{aligned} \tag{34}$$

$$\begin{aligned} &INV^{\langle x,y\rangle,d}(KC^{\langle x,y\rangle,d'}(\mathrm{Op}^{\langle x,y\rangle,d''}(\mathrm{G}))) \\ =&\,KC^{\langle x,y\rangle,d}(INV^{\langle x,y\rangle,d'}(\mathrm{Op}^{\langle x,y\rangle,d''}(\mathrm{G}))). \end{aligned} \tag{35}$$

$$\begin{aligned} &INV^{\langle x,y\rangle,d}\left(UNION^{\langle x,y\rangle,d'}(\mathrm{Op}^{\langle x,y\rangle,d_1}(\mathrm{G}_1), \mathrm{Op}^{\langle x,y\rangle,d_2}(\mathrm{G}_2))\right) \\ =&\,UNION^{\langle x,y\rangle,d}\Big(INV^{\langle x,y\rangle,d_1'}(\mathrm{Op}^{\langle x,y\rangle,d_1}(\mathrm{G}_1)), \\ &\,INV^{\langle x,y\rangle,d_2'}(\mathrm{Op}^{\langle x,y\rangle,d_2}(\mathrm{G}_2))\Big). \end{aligned} \tag{36}$$

INV is pushed down whenever possible to eventually reach SCAN, as inverse-label edges are typically stored in the database and can be retrieved directly.

4.1.2 Planning Algorithm. The input to Alg. 2, the main CRPQ planning algorithm, is a CRPQ query graph Q, and the output is the query plan $T(Q)$ chosen for it. Alg. 2 lazily invokes Alg. 1 to preprocess the RPQs in the CRPQ when necessary.

RPQ Preprocessing. The main preprocessing routine of Alg. 1, `rpqPreprocess`, first plans each subquery using `planSubRpq`, which delegates to `kleenePlanning` for Kleene closures, BPT-SCAN for single edge labels (and their inverses), and recursively invokes `rpqPreprocess` for all other cases. `kleenePlanning` constructs a CAKC plan that uses the nested RPQ's plan as its operand. When the source or target is a constant, CAKC handles it directly rather than pushing it down to the child plan, as described in Sec. 3.3. Once subqueries are planned, they are combined using MU for | (alternation) and passed to `crpqPlanning` for / (concatenation). Constant sources or targets are pushed down according to Eq. 29–32.

The input parameter *isINV* in the functions of Alg. 1 indicates whether the current subquery is wrapped by the INV operator. This arises from reversing the edge or path direction to meet TI constraints (Alg. 2, line 13). The flag is propagated unchanged through recursive calls, after which BPT-SCAN retrieves the appropriate inverse labels, as specified by Eq. 33–36.

CRPQ Planning. Alg. 2 follows a dynamic programming (DP) framework over the edges of the query subgraphs. For each edge set E_k with k edges, Q_k denotes the query subgraph induced by it. We can obtain Q_k from smaller query subgraphs via TIs or SJs. Among all possible ways to obtain Q_k, the one with the lowest estimated cost is selected and stored in the query plan cache `PlanMap`. During the DP, only connected query subgraphs are considered. The following are the recursive conditions used in the DP:

- **TI (lines 7–23):** The leftmost operand PlanMap[$Q_{k'}$] must have an edge set $E_{k'}$ that is a proper subset of E_k, and all edges in $E_k \setminus E_{k'}$ must share a common query vertex z, whether it be a constant or the intersected variable. The algorithm enumerates all $Q_{k'}$ that satisfy these conditions and estimates the cost of extending it with the edges in $E_k \setminus E_{k'}$ using an LFJ (line 18). The

Algorithm 1: RPQ Preprocessing in CRPQ Plan Selection

```
1  Function rpqPreprocess(e(x, y), isINV):
2      curOp ← the lowest-precedence operator in e
3      if curOp = | then
4          P ← empty plan
5          foreach RPQ subquery r separated by | do
6              if P is an empty plan then
7                  P ← planSubRpq(r(x, y), isINV)
8              else
9                  P ← MU(P, planSubRpq(r(x, y), isINV))
10         return P
11     else if curOp = / then
12         E_e ← ∅; rlist ← the list of RPQ subquery r's separated by /
13         foreach i ∈ {0, 1, ··· , rlist.length() − 1} do
14             if i = 0 then
15                 s ← x, t ← a new variable
16             else if i = rlist.length() − 1 then
17                 s ← t, t ← y
18             else
19                 s ← t, t ← a new variable
20             E_e ← E_e ∪ {planSubRpq(rlist[i](s, t), isINV)}
21         return crpqPlanning(E_e)
22     else
23         return planSubRpq(e(x, y), isINV)
24 Function kleenePlanning(r(x, y), isINV):
25     Suppose r = R+ or R*
26     if x is a constant then
27         Let z ∈ V be a new variable
28         curPlan ← rpqPreprocess(R(z, y), isINV)
29         return CAKC_{z↦x}(curPlan)
30     else if y is a constant then
31         Let z ∈ V be a new variable
32         curPlan ← rpqPreprocess(R(x, z), isINV)
33         return CAKC_{z↦y}(curPlan)
34     else
35         curPlan ← rpqPreprocess(R(x, y), isINV)
36         return CAKC(curPlan)
37 Function planSubRpq(r(x, y), isINV):
38     if r = R+ or r = R* then
39         P ← kleenePlanning(r(x, y), isINV)
40     else if r is not a single edge label then
41         P ← rpqPreprocess(r(x, y), isINV)
42     else
43         P ← BPT-SCAN(−r(x, y)) if isINV else BPT-SCAN(r(x, y))
44     return P
```

Algorithm 2: CRPQ Plan Enumeration & Selection

```
   Input: CRPQ Q
   Output: CRPQ Q's query plan T(Q)
1  Function crpqPlanning(q):
2      PlanMap ← empty cache
3      for k ∈ {2, 3, ··· , |E_Q|} do
4          foreach E_k ⊆ E_Q s.t. |E_k| = k and E_k is connected do
5              minCost ← ∞
6              Let Q_k be the query subgraph induced by E_k
               /* Find the most efficient plan using TI */
7              foreach E_k' s.t. E_k' is connected and ∃z ∈ V ∪ V s.t.
                 ∀e ∈ E_k \ E_k', e = e(x_e, z) ∨ e = e(z, x_e) do
8                  Let Q_k' be the query subgraph induced by E_k'
9                  curE ← ∅
10                 foreach e(z, x_e) ∈ E_k \ E_k' do
11                     if INV(e) not in PlanMap then
12                         PlanMap[INV(e)] ←
                             rpqPreprocess(e(x_e, z), 1)
13                     curE ← curE∪{PlanMap[INV(e)]}
14                 foreach e(x_e, z) ∈ E_k \ E_k' do
15                     if e not in PlanMap then
16                         PlanMap[e] ← rpqPreprocess(e(x_e, z), 0)
17                     curE ← curE∪{PlanMap[e]}
18                 P← TI(PlanMap[Q_k'], curE)
19                 if cost(P) < minCost then
20                     minCost ← cost(P); bestPlan ← P
               /* Find the most efficient plan using SJ */
21             foreach E_k1 ∪ E_k2 = E_k and E_k1, E_k2 are both connected do
22                 P ← SJ(PlanMap[Q_k1], PlanMap[Q_k2])
23                 if cost(P) < minCost then
24                     minCost ← cost(P); bestPlan ← P
25             PlanMap[Q_k] ← bestPlan
26     return PlanMap(Q)
```

plans of the query edges or paths are selected by invoking Alg. 1 on demand (lines 10-17). Specifically, to ensure that the traversed query edges or paths extend towards z according to Eq. 18, the INV operator is applied to some of them (lines 10-13).

- **SJ (lines 24–27):** For an SJ, the operands Q_{k_1} and Q_{k_2} must have edge sets E_{k_1} and E_{k_2} that form a partition of E_k. In addition, both E_{k_1} and E_{k_2} must induce connected query subgraphs. The algorithm enumerates all such valid pairs Q_{k_1} and Q_{k_2}, and estimates the cost of performing an HJ between them, relying on the selected plans for Q_{k_1} and Q_{k_2} already available in `PlanMap`.

Example 4.1. Fig. 13 shows three alternative plans considered during Alg. 2's planning process of the query in Fig. 1 when $k = 5$, so the subquery considered at this stage is the full CRPQ. When the data graph is Fig. 2, the most efficient plan is Fig. 13a as explained in Sec. 3.4.4, intuitively because its operands have lower cardinalities compared with Fig. 13b and 13c.

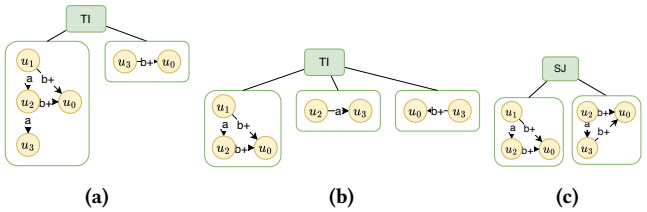


Figure 13: Example alternative plans considered when $k = 5$ in the planning process of the CRPQ in Fig. 1.

4.2 Cost & Cardinality Estimation

Query plan enumeration and selection (Alg. 1 and 2) require estimating the evaluation cost of physical operators. Because this cost depends on operand cardinality (i.e., intermediate result size), cardinality must be estimated first. In the following, we present cardinality estimation for logical operators and cost estimation for their physical counterparts.

4.2.1 TI. In this paper, we extend the cardinality estimation framework in Yang et al. [26]. The key extension lies in our focus on

the edge set of the query subgraph rather than its vertex set during cardinality estimation, because TI does not require all query edges adjacent to a query vertex to be processed simultaneously. Specifically, for each E_k in Alg. 2, we maintain several rows of sampled intermediate query results, which have been progressively expanded through TIs. In each expansion step, we use reservoir sampling to determine whether the expanded sample should be retained. As the sample for E_k is derived from that of E_{k-1}, the cardinality of E_k can be estimated as follows:

$$\mathrm{card}(E_k) \leftarrow \max(\mathrm{count} \times \mathrm{card}(E_{k-1})/\mathrm{S}(E_{k-1}), 1), \tag{37}$$

where $\mathrm{S}(E_{k-1})$ denotes the size of E_{k-1}'s sample set, and count refers to the number of rows in E_{k-1}'s sample set that successfully expands to form E_k. When TI has a constant source, the sampling procedure initiates from that constant. Constant targets are transformed into constant sources by Eq. 30.

We use LFJs to implement TIs. Let u be the LFJ operator node, and $w_1, \cdots, w_m$ be its child nodes.

LFJ's Joining and Sorting Cost. When the TI is n-ary, i.e, it has at least three operands, then for each hyperedge in $[\![w_1]\!]$, an LFJ is conducted on the ordered neighbor lists of the common vertices between the leftmost and the other operands to match the intersected query vertex. The i-th iteration of the LFJ takes $O(N_{\min,i} \log \frac{N_{\max,i}}{N_{\min,i}})$ time [23], where $N_{\min,i}$ and $N_{\max,i}$ are the lengths of the shortest and longest neighbor lists involved, respectively. We can thus derive an upper bound on the overall cost:

$$\begin{aligned}
\mathrm{cost}^{\mathrm{op}}_{\mathrm{LFJ},i}(u) &= \begin{cases} 0 & \text{if } N_{\min,i} = 0 \\ N_{\min,i} \log \frac{N_{\max,i}}{N_{\min,i}} & \text{otherwise} \end{cases} \\
\mathrm{cost}^{\mathrm{op}}_{\mathrm{LFJ}}(u) &= \sum_{i=1}^{|[\![w_1]\!]|} \mathrm{cost}^{\mathrm{op}}_{\mathrm{LFJ},i}(u) \leq \log(\max_{2\leq j\leq m} |[\![w_j]\!]|) \cdot \sum_{i=1}^{|[\![w_1]\!]|} N_{\min,i} \\
&\leq \min_{2\leq j\leq m} |[\![w_j]\!]| \cdot \log(\max_{2\leq j\leq m} |[\![w_j]\!]|).
\end{aligned} \tag{38}$$

In addition, all the neighbor lists involved in the LFJ need sorting, the cost of which is upper bounded as

$$\mathrm{cost}^{\mathrm{sort}}_{\mathrm{LFJ}}(u) \leq \sum_{j=2}^{m} |[\![w_j]\!]| \cdot \log |[\![w_j]\!]|. \tag{39}$$

When the TI is binary, no intersection or sorting is necessary, so $\mathrm{cost}^{\mathrm{op}}_{\mathrm{LFJ}}(u) = |[\![w_1]\!]|$ and $\mathrm{cost}^{\mathrm{sort}}_{\mathrm{LFJ}}(u) = 0$.

LFJ's Checking Cost. When the intersected query vertex is already in the leftmost operand of the TI, we need to additionally check if its match is in the output of the LFJ, which is implemented using a binary search, whose time cost is $O(\log N_{\mathrm{out},i})$, proportional to the length of the output vertex list in the i-th iteration. Since the LFJ essentially computes the intersection of the neighbor lists involved in each iteration, $\sum_{i=1}^{|[\![w_1]\!]|} N_{\mathrm{out},i} \leq \min_{2\leq i\leq m} |[\![w_i]\!]|$. Therefore, the overall checking cost is estimated as:

$$\mathrm{cost}^{\mathrm{check}}_{\mathrm{LFJ}}(u) = \sum_{i=1}^{|[\![w_1]\!]|} \log N_{\mathrm{out},i} \leq |[\![w_1]\!]| \log \frac{\min_{2\leq j\leq m} |[\![w_j]\!]|}{|[\![w_1]\!]|} \tag{40}$$

LFJ's Overall Cost. In summary, the overall cost of an LFJ is estimated as $\mathrm{cost}_{\mathrm{LFJ}}(u) = \mathrm{cost}^{\mathrm{op}}_{\mathrm{LFJ}}(u) + \mathrm{cost}^{\mathrm{sort}}_{\mathrm{LFJ}}(u) + \mathrm{cost}^{\mathrm{check}}_{\mathrm{LFJ}}(u)$.

4.2.2 SJ. Since cardinality estimation for an SJ is equivalent to estimating the number of matches of the corresponding query subgraph, and Alg. 2 considers TIs before SJs for the same query subgraph, we can perform cardinality estimation for TIs only and reuse the cached estimates when considering SJs.

We use HJs to implement SJs, whose cost consists of the cost of the join itself and building the hash table on an operand's results. Let u be an SJ operator node and let w_1 and w_2 be its operand nodes. The hash table is constructed on one of $[\![w_1]\!]$ and $[\![w_2]\!]$, and the other operand's results are iterated to probe the hash table, both incurring amortized $O(1)$ time for each hyperedge. Therefore, the total cost of an HJ is estimated as follows:

$$\mathrm{cost}_{\mathrm{HJ}}(u) = \mathrm{cost}^{\mathrm{op}}_{\mathrm{HJ}}(u) + \mathrm{cost}^{\mathrm{sort}}_{\mathrm{HJ}}(u) = |[\![w_1]\!]| + |[\![w_2]\!]|. \tag{41}$$

4.2.3 KC. We use the cardinality estimation method for Kleene closures proposed in Pang et al. [21]. Specifically, let u be the KC operator node that denotes R^+ or R^*, both of which have the same cardinality and cost due to the lazy handling of ϵ's (Sec. 3.3.4), its cardinality is estimated as follows:

$$\mathrm{card}(u) \leq \sum_{i=1}^{D} |[\![R^i]\!]| = \sum_{i=0}^{D-1} c^i |[\![R]\!]| = \frac{1-c^D}{1-c} |[\![R]\!]|, \tag{42}$$

where D is the maximum length of the regular path, that is, the number of iterations before the KC evaluation terminates, which is estimated as follows [21]; $c = \frac{|J_{l_1/R}| \cdot |[\![R]\!]|}{|[\![l_1]\!].t| \cdot |[\![R]\!].s|}$ represents the selectivity of the join operation, in which l_1 is the final edge label in R, $J_{l_1/R}$ is the set of data vertices that are both targets of l_1 and sources of R, estimated using the Monte Carlo sampling technique in Pang et al. [21], and $[\![r]\!].s$ and $[\![r]\!].t$ denote the set of source and target vertices in the results of r, respectively. If R ends with multiple possible edge labels due to alternation, c should be calculated separately for each possible final label and then summed to obtain the final selectivity estimate.

We use CAKCs to implement KCs, which joins the previous iteration's results with $[\![R]\!]$ for each iteration, whose cost is estimated by $\mathrm{cost}_{\mathrm{CAKC}}(u) = \mathrm{card}(u)$.

4.2.4 Union. Since union involves performing a union on the input result hypergraphs, its cardinality estimate is simply the sum of the input result hypergraphs' sizes; so is the cost estimate of MU.

4.2.5 INV. INV is never physically evaluated, but is instead pushed down according to the transformation rules until reaching SCAN. Therefore, it does not incur any cost and keeps the same cardinality.

4.2.6 SCAN. Both the cardinality and cost of SCAN equate to the number of edges that it will retrieve from the data graph.

5 EXPERIMENTAL EVALUATION

Dataset & workload. We use WDBench [3] and LDBC SNB [8] as the benchmarks. The statistics of the datasets are shown in Tab. 4. As for the workload, we use the CRPQ class from WDBench and the read-only queries from the SNB Interactive Workload. The WDBench subset comprises challenging CRPQs that historically required over 60 seconds to execute on the Wikidata SPARQL endpoint, as recorded in its query logs [16]. For the SNB workload, we exclude queries IS1 to IS6, as they lack query edges beyond single edge labels. Additionally, we adapt the remaining SNB queries by removing any constructs that are outside the scope of CRPQs and instantiate them using five sets of randomly sampled query parameters, yielding 80 queries in total.

Table 4: Dataset Statistics.

Dataset	# Vertices ($\|V\|$)	# Edges ($\|E\|$)	# Edge Labels ($\|\lambda\|$)
Wikidata (WDBench)	348,945,080	958,844,164	5,419
SF10 (LDBC SNB)	36,485,761	231,371,359	15

Table 5: Comparison with existing systems.

Benchmark	Method	Avg. E2E Query Time (s)	Avg. Speedup	Max Speedup	# Time-outs
WDBench	MillenniumDB	5.5544	—	—	186
	Ours	2.6662	2.0833	66703.3178	60
LDBC SNB	Neo4j	5.3256	—	—	10
	Ours	3.1638	1.6833	224.8514	10

(a) Total query time. **(b) Execution time.** **(c) Planning time.**

Figure 14: Violin plots of MillenniumDB's & our runtimes.

Environment. We conduct all our experiments on a machine with 96 Intel Xeon 2.1GHz CPUs and 1TB RAM. Both of the compared methods are implemented in C++ and executed with a single thread.

Baseline. We choose MillenniumDB [1] and Neo4j [18] as the basis for implementing our method. We implement our operators and the planning procedure directly on top of the storage layer in MillenniumDB and by utilizing the user-defined procedure APIs in Neo4j. The plan space of MillenniumDB and Neo4j is strictly subsumed by ours; their operator support and planning strategy are discussed in Sec. 6.

5.1 Overall Comparison

In this section, we compare the overall efficiency of our approach against MillenniumDB and Neo4j on the CRPQs in WDBench and LDBC SNB. We set a 60-second timeout on the end-to-end (E2E) query time, i.e., the sum of the query planning and execution time.

Tab. 5 shows the overall runtime statistics. Our approach achieves a significant average speedup of 2.08× and a maximum speedup of over 4 orders of magnitude over MillenniumDB in terms of E2E query time, underscoring the effectiveness of our proposed algebraic optimizations. In addition, our approach reduces the number of timeouts by approximately 68%, which clearly indicates improved robustness. Fig. 14a corroborates this finding, demonstrating that our approach not only has a notably lower median E2E query time, but also exhibits a tighter range compared to MillenniumDB, suggesting greater consistency in performance. Our speedup compared with Neo4j on LDBC SNB (1.68×) is notable, which demonstrates the robustness of our approach over diverse data and query sets. Though the speedup is not as high as that compared with MillenniumDB on WDBench, it must be weighed against the fact that Neo4j's query language, Cypher, lacks full support for CRPQs, as it only allows Kleene closures of single edge label unions rather than complete RPQ syntax and semantics.

To understand the source of the overall speedup, we show the decomposition of the E2E time on WDBench into its constituent components, execution time and planning time, in Fig. 14b and 14c. The majority of the performance gain stems from the time-consuming execution phase, as the planning times of both systems are comparable. The significant variance in execution times is primarily driven by differences in query complexity and the resulting search space size on the data graph. To confirm this, we examined the queries in WDBench that successfully executed but had a total time exceeding 55 seconds. The base MillenniumDB recorded five such queries, compared to only one for our approach. All of these queries are highly complex: they contain at least four RPQs, at most two constant endpoints, at least one Kleene closure, and involve at least one edge label with a cardinality exceeding one million. The time decomposition on LDBC SNB shows a similar trend and the figures are omitted due to limited space.

5.2 Varying Query Complexity

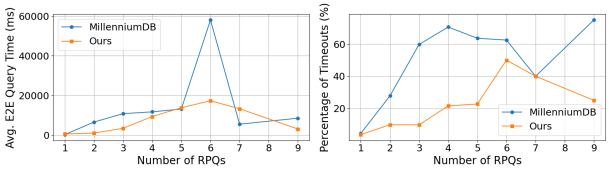
In this section, we evaluate the performance of our approach against MillenniumDB across four dimensions of CRPQ complexity: the number of RPQs and the total number of edge labels, Kleene closures, and constant vertices across RPQs. The complexity of CRPQs generally increases with the number of RPQs, edge labels, and Kleene closures and decreases with the number of constant vertices. Each subplot in Fig. 15 reports the average E2E query time and the percentage of timeouts with varying query complexity. The plots do not exhibit linear growth or decay but rather fluctuations due to the interplay among the multiple factors influencing complexity.

Our method consistently outperforms MillenniumDB in both efficiency and robustness across nearly all tested complexity levels, achieving significantly lower E2E query time and fewer timeouts overall. Critically, our approach can successfully evaluate challenging query classes where MDB fails completely (100% timeouts), including workloads with 9 RPQs and 11 edge labels. This establishes our method's capability to reliably process high-complexity CRPQs within practical latency bounds.

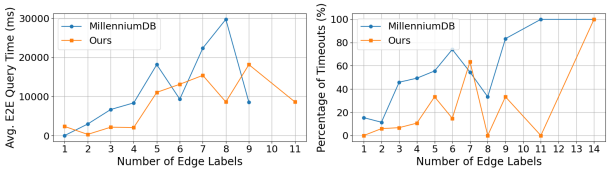
6 RELATED WORK

CRPQ Evaluation. Existing research and DBMSs do not offer a systematic framework for CRPQ evaluation that combines a unified plan space, algebraic-rewriting-based plan enumeration, and cost-based selection, as shown in Tab. 1. Specifically:

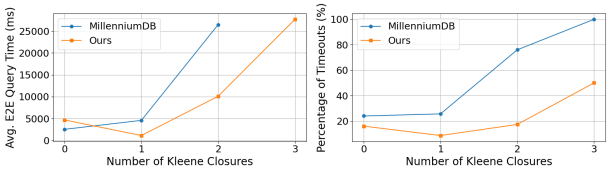
- **Join-Ordering** [2] iteratively joins query edges using a join operator equivalent to binary TIs but processes RPQs separately using the ALP procedure [11]. Then, the join orders of query edges are enumerated as plans using dynamic programming. Its sampling-based cost model estimates RPQ join costs by assuming unbounded target vertices.
- **MWJ-CRPQ** [14] offers a de-facto unified plan space for CRPQs using a multi-way join operator adapted from Generic Join [20]



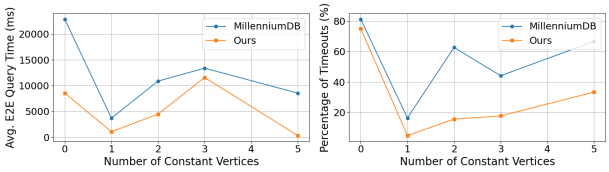
(a) Varying # RPQs.



(b) Varying # edge labels.



(c) Varying # Kleene closures.



(d) Varying # constant vertices.

Figure 15: Comparison between MillenniumDB & our method on CRPQs with varying complexity.

and a Kleene closure operator. However, it relies on a greedy dynamic variable ordering guided by sampling-based cost estimates, which isolates nested RPQ components into separate layers, ignoring their interplay and risking local optima.

- **Neo4j** [18] and **MillenniumDB** [24] employ separate operators for single-label edges, RPQs, and their conjunctions. Their RPQ plans are fixed as DFAs. Notably, Neo4j does not support multi-way joins, while MillenniumDB only uses multi-way joins for single-label query edges.

Subgraph Matching. Existing subgraph matching evaluation methods predominantly adopt a backtracking-based search paradigm with the filtering-ordering-enumeration framework [29], which is equivalent to query plans that exclusively use TI operators while mandating that all query edges incident to a given vertex are processed with one operator. Such constraints align with the worst-case optimal join (WCOJ) paradigm [19], which employs multi-way joins to ensure the total evaluation time is bounded by the AGM bound, a worst-case upper bound on the output size relative to input relations. However, for acyclic query graphs, classical binary join (BJ) algorithms, e.g., Yannakakis' [28], achieve time complexity no worse than $O(AGM)$. Consequently, neither WCOJ nor BJ universally dominates the other across diverse query structures and data distributions. This observation motivates hybrid query processing frameworks [17] that dynamically integrate WCOJ and BJ operators during plan enumeration and selection. Our approach offers even greater flexibility by removing the constraint of WCOJ that query edges incident to a query vertex need to be handled together.

RPQ Evaluation. Regular path query (RPQ) evaluation methods can be broadly categorized into two paradigms, automaton-based and extended relational algebra (RA)-based. The automaton-based approach compiles the regular expression into a finite state automaton and traverses the data graph via breadth-first or depth-first search guided by automaton transitions. Data vertices reached at the initial and accept states are recorded as result pairs. In contrast, the extended RA-based approach extends traditional RA with specialized operators to handle Kleene closures. For example, Fejza et al. [9] extends RA with a fix-point operator to express recursion algebraically and proposes a compositional optimizer that applies rewrite rules and cost-based plan selection.

In the absence of Kleene closures, automaton-based evaluation is equivalent to left-deep TI plans, while the RA-based approach supports both left-deep and bushy TI plans. When Kleene closures are present, a fundamental distinction emerges: automaton-based methods process closures through iterative traversal without caching intermediate results, while RA-based methods explicitly cache the result of R and compute its fixpoint iteratively for R^+ or R^*. A partial loop caching strategy is further proposed to cache results of R's subqueries [25]. In our framework, such strategies correspond to distinct physical implementations of the KC operator.

7 CONCLUSION & FUTURE WORK

This work addresses a fundamental gap in graph database systems: the absence of a unified optimization framework for conjunctive regular path queries (CRPQs). By recognizing that existing approaches fuse subgraph matching and RPQ components using heuristic strategies that yield sub-optimal performance, we propose a cohesive algebraic framework grounded in hypergraph-based composability. Our core contributions include: a hypergraph query model that overcomes limitations in ISO/IEC-standardized languages (e.g., GQL and SQL/PGQ); algebraic operators that abstract CRPQ semantics and bridge the representation gap between tree-shaped join plans and automata-based path evaluations; a plan enumeration scheme based on algebraic transformation rules and a cost-based optimizer for efficient plan selection; and implementations on open-source graph database systems validated by substantial performance gains. Future work will explore more efficient physical implementations on the algebraic basis and novel hypergraph analytical algorithms that leverage the composability of our query model.

ACKNOWLEDGMENTS

Pang and Zou's work was supported by NSFC under grant 62532001. Bonifati's work was partially supported by the ERC AdG grant GOY nr. 101199575. Özsu's work was supported by a grant from the NSERC of Canada. Pang and Zhou's work was supported by Hong Kong Research Grants Council (grant#R6005-25). This work was partially conducted in JC STEM Lab of Data Science Foundations funded by The Hong Kong Jockey Club Charities Trust. Lei Zou is the corresponding author of this work.

REFERENCES

[1] [n.d.]. MillenniumDB. https://github.com/MillenniumDB/MillenniumDB Accessed: 2026-03-08.

[2] Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Minh-Hoang Dang, and Brice Nédelec. 2023. Join Ordering of SPARQL Property Path Queries. In *The Semantic Web*, Catia Pesquita, Ernesto Jimenez-Ruiz, Jamie McCusker, Daniel Faria, Mauro Dragoni, Anastasia Dimou, Raphael Troncy, and Sven Hertling (Eds.). Vol. 13870. Springer Nature Switzerland, Cham, 38–54. https://doi.org/10.1007/978-3-031-33455-9_3

[3] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. WDBench: A Wikidata Graph Query Benchmark. In *The Semantic Web – ISWC 2022*, Ulrike Sattler, Aidan Hogan, Maria Keet, Valentina Presutti, João Paulo A. Almeida, Hideaki Takeda, Pierre Monnin, Giuseppe Pirrò, and Claudia d'Amato (Eds.). Vol. 13489. Springer International Publishing, Cham, 714–731. https://doi.org/10.1007/978-3-031-19433-7_41

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages. https://doi.org/10.1145/3104031

[5] Renzo Angles, Angela Bonifati, Roberto García, and Domagoj Vrgoč. 2025. Path-Based Algebraic Foundations of Graph Query Languages. https://doi.org/10.48786/EDBT.2025.63

[6] Katerina Böhmová, Jérémie Chalopin, Matús Mihalák, Guido Proietti, and Peter Widmayer. 2016. *Sequence Hypergraphs*. Springer, United States, 282–294. https://doi.org/10.1007/978-3-662-53536-3_24

[7] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Synthesis Lectures on Data Management, Vol. 10. Morgan & Claypool Publishers. 1–184 pages. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[8] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. https://doi.org/10.1145/2723372.2742786

[9] Amela Fejza, Pierre Genevès, Nabil Layaïda, and Sarah Chlyah. 2023. The µ-RA System for Recursive Path Queries over Graphs. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management* (Birmingham, United Kingdom) *(CIKM '23)*. Association for Computing Machinery, New York, NY, USA, 5041–5045. https://doi.org/10.1145/3583780.3614756

[10] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *Proc. VLDB Endow.* 18, 6 (Aug. 2025), 1798–1810. https://doi.org/10.14778/3725688.3725707

[11] Steve Harris and Andy Seaborne. 2013. *SPARQL 1.1 Query Language*. W3C Recommendation. World Wide Web Consortium (W3C). https://www.w3.org/TR/sparql11-query/ Accessed: 2026-03-08.

[12] ISO. 2023. ISO/IEC 9075-16:2023 Information technology — Database languages SQL - Part 16: Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html. Accessed: 2026-03-08.

[13] ISO. 2024. ISO/IEC 39075:2024 Information technology — Database languages — GQL. https://www.iso.org/standard/76120.html. Accessed: 2026-03-08.

[14] Nikolaos Karalis, Alexander Bigerl, Liss Heidrich, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo. 2024. Efficient Evaluation of Conjunctive Regular Path Queries Using Multi-way Joins. In *The Semantic Web*, Albert Meroño Peñuela, Anastasia Dimou, Raphaël Troncy, Olaf Hartig, Maribel Acosta, Mehwish Alam, Heiko Paulheim, and Pasquale Lisena (Eds.). Springer Nature Switzerland, Cham, 218–235.

[15] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. 2021. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 925–937. https://doi.org/10.1145/3448016.3457265

[16] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *The Semantic Web – ISWC 2018*, Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl (Eds.). Springer International Publishing, Cham, 376–394.

[17] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[18] Neo4j, Inc. 2026. Neo4j. https://neo4j.com. Accessed: 2026-03-08.

[19] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Scottsdale, Arizona, USA) *(PODS '12)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2213556.2213565

[20] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16. https://doi.org/10.1145/2590989.2590991

[21] Yue Pang, Lei Zou, Jeffrey Xu Yu, and Linglin Yang. 2024. Materialized View Selection & View-Based Query Planning for Regular Path Queries. *Proc. ACM Manag. Data* 2, 3, Article 152 (May 2024), 26 pages. https://doi.org/10.1145/3654955

[22] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *The VLDB Journal* 29, 2-3 (May 2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[23] Todd L Veldhuizen. 2014. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*.

[24] Domagoj Vrgoč, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2023. MillenniumDB: An Open-Source Graph Database System. *Data Intelligence* (June 2023), 1–39. https://doi.org/10.1162/dint_a_00209

[25] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query Planning for Evaluating SPARQL Property Paths. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco California USA, 1875–1889. https://doi.org/10.1145/2882903.2882944

[26] Linglin Yang, Lei Yang, Yue Pang, and Lei Zou. 2022. gCBO: A Cost-based Optimizer for Graph Databases. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 5054–5058.

[27] Rongjian Yang, Zhijie Zhang, Weiguo Zheng, and Jeffrey Xu Yu. 2023. Fast Continuous Subgraph Matching over Streaming Graphs via Backtracking Reduction. *Proc. ACM Manag. Data* 1, 1, Article 15 (May 2023), 26 pages. https://doi.org/10.1145/3588695

[28] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7* (Cannes, France) *(VLDB '81)*. VLDB Endowment, 82–94.

[29] Zhijie Zhang, Yujie Lu, Weiguo Zheng, and Xuemin Lin. 2024. A Comprehensive Survey and Experimental Study of Subgraph Matching: Trends, Unbiasedness, and Interaction. *Proceedings of the ACM on Management of Data* 2, 1 (March 2024), 1–29. https://doi.org/10.1145/3639315