Materialized View Selection & View-Based Query Planning for Regular Path Queries

YUE PANG, Peking University, China
LEI ZOU, Peking University, China
JEFFREY XU YU, Chinese University of Hong Kong, China
LINGLIN YANG, Peking University, China

A regular path query (RPQ) returns node pairs connected by a path whose edge label sequence satisfies the given regular expression. Given a workload of RPQs, selecting the shared subqueries as materialized views to precompute offline can speed up the online processing. Since the available memory is limited, we define the materialized view selection (MVS) problem for RPQs as minimizing the total workload query cost within a memory budget. To tackle the problem's NP-hardness, we design an efficient MVS algorithm based on heuristics. To prevent redundancies in the selected views, we devise the AND-OR directed acyclic graph with closure (AODC) as the multi-RPQ query plan representation for the workload, which encodes the relations between subqueries. In addition to detecting view redundancy, the AODC also incrementally updates itself during view selection. To support query planning, we design a scalable cost and cardinality estimation scheme for full-fledged RPQs, including Kleene closures. Our method, when applied to the Wikidata Query Logs, shows a 9.73× speedup in the total query processing time compared to ad-hoc processing, using the views it selects.

CCS Concepts: • **Information systems** → **Query planning**; **Database views**.

Additional Key Words and Phrases: regular path query, materialized view, query planning

ACM Reference Format:
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 (SIGMOD), Article 152 (June 2024), 26 pages. https://doi.org/10.1145/3654955

1 INTRODUCTION

A regular path query (RPQ) aims to return all the node pairs with a path satisfying the given regular expression on the set of edge labels in the graph. It can thus help retrieve node pairs that are complexly related. For example, if we want to get all the users whose direct or indirect acquaintances moderate a forum in a social network, where the users and forums are nodes and `knows` and `moderates` are edge labels, we can write the following RPQ: $\texttt{knows}^+/\texttt{moderates}$, which matches paths with one or more `knows` edges followed by a `moderates` edge, returning the desired user-forum pairs. Due to their wide usage in real applications, RPQs are at least partially supported

Authors' addresses: Yue Pang, michelle.py@pku.edu.cn, Peking University, Beijing, China; Lei Zou, zoulei@pku.edu.cn, Peking University, Beijing, China; Jeffrey Xu Yu, yu@se.cuhk.edu.hk, Chinese University of Hong Kong, Hong Kong, China; Linglin Yang, linglinyang@stu.pku.edu.cn, Peking University, Beijing, China.

ACM 2836-6573/2024/6-ART152
https://doi.org/10.1145/3654955

in the most widely used graph query languages, including SPARQL[1], openCypher [11], and ISO/GQL [9]. More discussions of RPQ support in these languages will be given in Sec. 2.

Existing works on RPQs primarily study how to speed up an ad-hoc RPQ [27]. However, hardly any of them studies the optimization of an RPQ workload, i.e., a set of RPQs. One possible way to speed up the execution of an RPQ workload is to use materialized views. Given an RPQ workload, if we select some subqueries as materialized views and precompute their results offline, we can utilize them to accelerate online query processing.

Many applications need to efficiently process an RPQ workload, i.e., a set of RPQs, categorized as follows:

- Applications with fixed RPQ workloads: Certain highly specialized applications, e.g., signaling pathway detection in protein interaction networks [22], have limited sets of domain-specific meaningful RPQs, which can be treated as fixed workloads.
- Applications with query logs involving RPQs: public graph query services, e.g., SPARQL query endpoints, commonly preserve history queries in the form of query logs, where similar queries occur over time in streaks [8]. Thus, extracting the RPQs from the query log as a workload speeds up similar RPQs in the future.
- Applications issuing multiple RPQs simultaneously: For example, visual query systems fuzzily translate a graphical query representation into multiple similar RPQs [1], forming a workload.

Materializing every RPQ in the workload will lead to the highest online efficiency if it is possible to do so. However, the available amount of memory may not be sufficient for storing the results of all the workload RPQs. Thus, it is meaningful to study materialized view selection for RPQs with the objective of minimizing the total query cost of the workload within a memory budget.

The materialized view selection problem for RPQs is NP-hard (Sec. 5.1). The hardness is intuitively due to the exponential number of view combinations concerning the workload size. It is thus difficult to compute the optimal set of materialized views in a reasonable amount of time. Therefore, we aim to design an efficient heuristic materialized view selection algorithm that returns a good enough view set. However, a heuristic selection scheme may result in many redundant views, which wastes memory without improving the efficiency, if it does not consider the relations between subqueries. In addition, the relations between subqueries are more complex than the string subsumption relation between their regular expressions, requiring a novel data structure to represent them.

To account for the relations between subqueries, we non-trivially extend the AND-OR directed acyclic graph (AND-OR DAG) [20], the plan representation for multi-query optimization in relational databases, which represents subquery relations as edges between subquery nodes. To adapt the AND-OR DAG for RPQs, we extend it with Kleene closure operators and call the resulting plan representation the AND-OR DAG with closure (AODC). We use the AODC to represent the joint plan of an RPQ workload. To support query planning with the AODC, we devise a cost and cardinality estimation scheme for RPQs that extends existing work by supporting Kleene closures. On this basis, we design a greedy materialized view selection algorithm with AODC-based redundancy detection and removal. We also design an incremental planning algorithm that updates the AODC with the selection of materialized views, so by the end of the view selection process, the optimal plans of all the workload queries using these views are ready for execution.

To sum up, our work makes the following contributions:

- We are the first to formally define and propose a principled approach for selecting materialized views for RPQs that minimizes the query cost of a given workload within a memory budget;
- We propose the AND-OR directed acyclic graph with closure (AODC) for multi-RPQ plan representation to detect and remove redundant views during materialized view selection;

[1] https://www.w3.org/TR/sparql11-query/

Table 1. Notations.

Notation	Description
$G = (V, E, \Sigma, l)$	An edge-labeled directed graph
Σ	The set of edge labels
l	The labeling function where $\forall e, l(e) \in \Sigma$
R	A regular path query (RPQ)
$[\![R]\!]_G$, $[\![R]\!]_G.s$, $[\![R]\!]_G.t$	The set of result node pairs of the RPQ R on the graph G, its set of source and target nodes
$\mathrm{v} = \langle R, [\![R]\!]_G \rangle$	A materialized view for the RPQ R
$\mathbb{V}$	A set of materialized views

Table 2. RPQ support in graph query languages.

Graph query language	SPARQL	openCypher	ISO/GQL
RPQ support level	Full	LCR	Full
Graph data model	RDF	PG	PG

- We design a scalable cost and cardinality estimation scheme for full-fledged RPQs, including Kleene closures;
- We conduct extensive experiments on the Wikidata dataset and a collection of challenging queries from the Wikidata Query Log, showing the the efficiency of our method. Specifically, the speedup on the workload is 9.73× with than without the materialized views, which use 2.14× memory of the data graph.

2 PROBLEM STATEMENT

Definition 1 (RPQ)**.** An RPQ R has the following form:

$$R \rightarrow \epsilon \mid a \mid a- \mid R_1/R_2 \mid R_1|R_2 \mid R? \mid R^* \mid R^+ \tag{1}$$

where 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, and $/, |, ?, *, +$ 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's result on the edge-labeled directed graph $G = (V, E, \Sigma, l)$ is a set of node pairs, recursively defined as follows:

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

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

$$[\![R_1/R_2]\!]_G = [\![R_1]\!]_G \bowtie_{[\![R_1]\!]_G.t=[\![R_2]\!]_G.s} [\![R_2]\!]_G \text{ (Concatenation)} \tag{4}$$

$$[\![R_1|R_2]\!]_G = [\![R_1]\!]_G \cup [\![R_2]\!]_G \text{ (Alternation)} \tag{5}$$

$$[\![R?]\!]_G = \{\langle v, v\rangle | v \in V\} \cup [\![R]\!]_G \text{ (Zero-or-one)} \tag{6}$$

$$[\![R^+]\!]_G = \cup_{i=1}^{\infty} [\![R^i]\!]_G \text{ (Kleene plus)} \tag{7}$$

$$[\![R^*]\!]_G = \{\langle v, v\rangle | v \in V\} \cup [\![R^+]\!]_G \text{ (Kleene star)} \tag{8}$$

where the join in Eqn. 4 treats the sets of node pairs as two-column tables, $[\![R]\!]_G.s$ and $[\![R]\!]_G.t$ referring to $[\![R]\!]_G$'s source and target nodes, respectively; R^k is the shorthand for concatenating R with itself for k times (e.g., $R^2 = R/R$). For an example of obtaining an RPQ's result on a graph, please refer to Sec. 5.4.

Definition 3 (Materialized View for RPQ)**.** Given an edge-labeled directed graph G, any tuple $\langle R, [\![R]\!]_G\rangle$ is a materialized view for RPQ, where R is called its query and $[\![R]\!]_G$ is called its result.

Definition 4 (RPQ Workload)**.** An RPQ workload $S = \{R_1, \cdots, R_k\}$ is a finite set of RPQs. To represent duplicate RPQs, we denote the RPQ R_i's frequency as freq$[R_i]$.

Definition 5 (Materialized View Selection (MVS) for RPQ)**.** Given an edge-labeled directed graph G, an RPQ workload S and a memory budget b, find the set of materialized views V that minimizes $\sum_{R \in S} cost(R, \mathsf{V})$ under the constraint $\sum_{\langle R, [\![R]\!]_G \rangle \in \mathsf{V}} |[\![R]\!]_G| < b$, where $cost(R, \mathsf{V})$ is the time cost of executing R with V.

RPQ support in graph query languages. Many graph query languages provides support for RPQ, but not all of them support RPQ fully (Def. 1). Tab. 2 shows the levels of RPQ support in three prominent graph query languages as examples. SPARQL and ISO/GQL fully supports RPQ, while openCypher only supports a fragment of RPQ called label-constrained reachability (LCR) [23], which disallows the concatenation operator. Since our method targets full-fledged RPQs, it supports the RPQ components in all these query languages, regardless of the graph data models they rely on.

3 METHOD OVERVIEW

As shown in Fig. 1, our method consists of two workflows: 1) query planning (on the left) and 2) materialized view selection (on the right). Given an RPQ workload and an edge-labeled directed data graph, the planner produces the multi-query plan for the workload (Sec. 5.3, Alg. 1). Then the view selector chooses views for materialization to minimize the workload's total query cost, with the multi-query plan aiding it to detect and remove redundant views (Sec. 6.1, Alg. 4). During view selection, the multi-query plan is incrementally updated accordingly to reflect how the query execution should utilize the selected views (Sec. 6.2, Alg. 5). Finally, the executor executes a workload query, using the materialized views according to the plan (Sec. 5.4, Alg. 2).

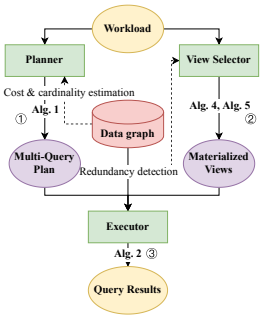


Fig. 1. Method overview.

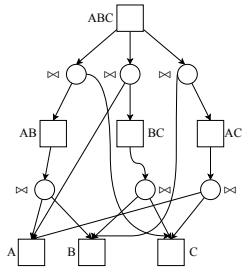


Fig. 2. An example AND-OR DAG.

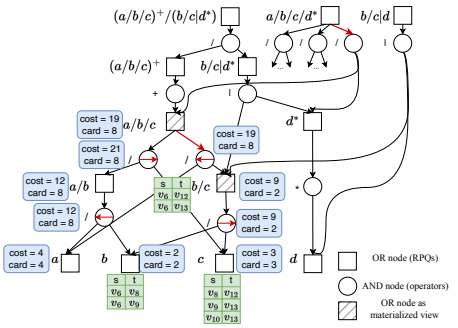


Fig. 3. An example AND-OR DAG with closure (AODC).

4 RELATED WORK

MVS for RPQs. To our knowledge, the only work that studies MVS for RPQs is [3]. Its method is not suitable for our setting, because their optimization objective is to minimize the memory usage, not the workload's query cost. Since the amount of memory that any materialized view uses is independent of each other, while a workload query's cost depends on whether other related queries are materialized, Afonin's [3] method is not directly adaptable to our setting. Besides, Afonin's [3] method exhaustively enumerates all the possible materialized view combinations without pruning, which is infeasible even on small workloads. Therefore, we do not compare our method with theirs experimentally.

Query planning for RPQs. Previous works on query planning for RPQs focus on optimizing ad-hoc RPQs. The state-of-the-art plan representation for RPQs is the Waveplan, used by the RPQ

optimizer Waveguide [27]. Waveplans extend finite automata in the following aspects: 1) using both forward and backward traversals on the data graph (e.g., $a\cdot$ signifies traversing a-labeled edges from their sources to their targets, while $\cdot a$ signifies traversing them from their targets to their sources); 2) allowing an RPQ plan to consist of multiple automata; 3) allowing views (i.e., the results of the other automata in the plan) as state transitions. With these extensions, Waveplans subsume the previous works in RPQ query planning, such as [13] which starts searching from the rare labels in the data graph. [27] also proposes a cost model to assess the efficiency of Waveplans. Since Waveplans can treat views as state transitions, it is possible to choose the most efficient Waveplan using a given set of materialized views. However, [27] does not provide a method to determine which views to select given an RPQ workload.

We use a plan representation different from Waveplans. We will explain why and contrast our plan space with theirs in Sec. 5.5.

In addition, some works translate RPQs into SQL extended with Kleene closures or Datalog to utilize their existing query planning mechanisms [10], but they ignore the specific graph query structure. [18] proposes a cost model for RPQs, which only applies to finite automaton plans. It is thus used for evaluating parallel RPQ plans, i.e., the ways of splitting an RPQ into a batch of subqueries or an RPQ workload into several batches for parallel execution, where a batch's cost is defined as the cost of its most expensive RPQ. [5] discusses RPQ execution on a space-efficient graph storage structure based on wavelet trees, outperforming existing graph DBMS on ad-hoc RPQs, so we also evaluate it experimentally. Other works discuss query planning for RPQs for distributed [19, 26] or approximate [25] query processing, which are beyond the scope of this paper.

Multi-query optimization (MQO) for RPQs. MQO aims to optimize a batch of queries by sharing computation. MQO and MVS are similar in that they both aim to optimize multiple queries at once. However, the difference in their settings leads to diverging optimization objectives. Specifically, MQO receives ad-hoc query batches, while MVS has a fixed query workload. Thus, MVS has an offline precomputation phase, where the time overhead is relatively unimportant within a reasonable range, while MQO does not.

Swarmguide [1, 2] is an MQO framework based on Waveguide [27]. Given an RPQ workload, Swarmguide clusters the workload queries into batches based on the similarity between their minimal deterministic finite automata (DFAs), which are unique up to isomorphism, using affinity propagation [12] on their labels. It then computes the maximum common sub-automaton of the minimal DFAs in each batch, selects that subquery as the batch's shared view, and uses Waveguide to plan it. After executing the shared views, Swarmguide uses them as state transitions and invokes Waveguide to plan the workload queries.

RTC [15] is an MQO framework with strategies to speed up the computation of the common Kleene closures R^* or R^+ in the given RPQ workload. Since these strategies are also effective on a single RPQ with Kleene closures according to their experiments, we consider RTC as an efficient *physical* implementation of the Kleene closure operator. In contrast, we and Waveguide [27] are primarily concerned with *logical* query planning, i.e., the operators' ordering, and our methods allows plugging in different physical implementations of the operators. Therefore, it is possible to incorporate RTC as an add-on to our query plan in the future work.

In our experiments, we compare these MQO methods with our MVS scheme fairly by moving their time overheads for extracting and conducting the shared computation to the offline phase, focusing on comparing their online query efficiency (Sec. 8.6).

MQO for relational queries. The AND-OR directed acyclic graph (AND-OR DAG), first proposed in [20], facilitates MQO in relational databases by representing the alternative plans of multiple queries in a single DAG. [21] proposes MVS algorithms for relational queries based on the AND-OR

DAG. Since the AND-OR DAG serves as our basis for MVS for RPQs, we briefly introduce it in the following.

The AND-OR DAG has two types of nodes: AND nodes, representing algebraic operators such as joins, and OR nodes, representing subqueries. AND nodes and OR nodes are interleaved by layer in the DAG: an OR node's out-neighbors can only be AND nodes, denoting alternative ways to compute the subquery's results, while an AND node's out-neighbors can only be OR nodes, denoting the operator's operands. For example, the AND-OR DAG in Fig. 2 shows all the possible ways to join the tables A, B, and C, drawing the OR nodes as squares and the AND nodes as circles. It subsumes all the alternative query plans of any workload that is a subset of $\{ABC, AB, BC, AC, A, B, C\}$. When an OR node has multiple child AND nodes (e.g., ABC), the cheapest child is selected for execution based on the cost model.

5 AND-OR DAG WITH CLOSURE

In this section, we describe the AND-OR DAG with closure (AODC), the multi-RPQ plan representation that we propose as the basis for MVS and view-based query planning.

5.1 Motivation

The motivation for designing the AODC for MVS and view-based query planning is: firstly, the hardness of MVS necessitates a heuristic MVS algorithm instead of exhaustive search; secondly, such a heuristic algorithm will lead to redundant views unless it considers the relations between the views, which is where the AODC comes into play. In the following, we first prove the hardness of MVS:

Theorem 1. Given a workload, MVS for RPQ is NP-hard.

Proof. Given a finite set N, if a function $f : 2^N \rightarrow \mathbb{R}$ only gives non-negative value and is monotone and submodular, finding a set $\mathsf{V} \subseteq N$ that maximizes $f(\mathsf{V})$ is NP-hard [16]. Given a workload S, we define $f(\mathsf{V})$ as the materialized view set V's benefit, i.e., cost reduction, where $\mathsf{V} \subseteq N$ and N is the set of the workload RPQs' subqueries. f accounts for the memory budget by ensuring $f(\mathsf{V} \cup \{\mathsf{v}\}) = f(\mathsf{V})$ if $\mathsf{V} \cup \{\mathsf{v}\}$ exceeds the budget. Thus, finding $\mathsf{V} \subseteq N$ to maximize $f(\mathsf{V})$ is equivalent to MVS for RPQ (Def. 5). If f has all of the three properties above, MVS for RPQ is NP-hard.

f obviously takes non-negative value only. For any view set V and view v, $f(\mathsf{V} \cup \{\mathsf{v}\}) \geq f(\mathsf{V})$, since if v does not reduce the workload's query cost, it will not be used. f is also submodular, i.e., for any view set V and U where $\mathsf{V} \subseteq \mathsf{U} \subseteq N$ and any view v, $f(\mathsf{V} \cup \{\mathsf{v}\}) - f(\mathsf{V}) \geq f(\mathsf{U} \cup \{\mathsf{v}\}) - f(\mathsf{U})$. For brevity, we denote $f(\mathsf{V} \cup \{\mathsf{v}\}) - f(\mathsf{V})$ by $\Delta_{\mathsf{V},\mathsf{v}}$ in the following. If neither V nor U's queries has subqueries that overlap with v's query, $\Delta_{\mathsf{V},\mathsf{v}} = \Delta_{\mathsf{U},\mathsf{v}}$. Otherwise, we denote the set of overlapping subqueries in V's queries as V' and those in U's queries as U'. Since $\mathsf{V} \subseteq \mathsf{U}$, $\mathsf{V}' \subseteq \mathsf{U}'$. When $\mathsf{V}' = \mathsf{U}'$, $\Delta_{\mathsf{V},\mathsf{v}} = \Delta_{\mathsf{U},\mathsf{v}}$; otherwise, the subqueries in $(\mathsf{U}' - \mathsf{V}')$'s queries can possibly reduce the workload's query cost further when materialized in place of v, so $\Delta_{\mathsf{V},\mathsf{v}} \geq \Delta_{\mathsf{U},\mathsf{v}}$. In all the possible cases, $\Delta_{\mathsf{V},\mathsf{v}} \geq \Delta_{\mathsf{U},\mathsf{v}}$, so f is submodular. Therefore, MVS for RPQ is NP-hard. □

Since MVS for RPQs is hard (Thm. 1), we have to resort to heuristic selection if we want to efficiently obtain a near-optimal solution, e.g., greedily selecting the top-ranking candidate views by some heuristic ordering until the memory exceeds the budget. However, such a selection scheme may lead to many redundant views if it does not consider the overlapping relations between the candidate views, which are more complicated than string subsumption. For example, for the workload query $(a/b/c)^+/(b/c|d^*)$, if $a/b/c$ is already materialized, the view a/b would be redundant, but b/c would not, though the strings a/b and b/c are both subsumed by $a/b/c$.

Hence, we adapt the AND-OR DAG, which originally represents the joint query plan of multiple relational algebra (RA) queries, for representing the joint query plan of the RPQ workload and the relations between candidate RPQ views. We call the resulting plan representation the AND-OR DAG with Closure (AODC).

5.2 Structure & Construction

We extend the AND-OR DAG with the Kleene closure operators, $*$ and +, and call the resulting plan representation the AND-OR DAG with closure (AODC), defined as follows:

Definition 6 (AND-OR DAG with Closure (AODC))**.** The AODC of the RPQ workload S is a labeled directed acyclic graph with two types of nodes, AND nodes and OR nodes. Since the AODC is acyclic, we hereby refer to any of its node's in- an out-neighbors as its parents and children, respectively.

An AODC satisfies the following properties:

- Each AND node is labeled by an RPQ opeator (i.e., /, |, ?, *, or +).
- Each OR node is labeled by a distinct RPQ. Those labeled by RPQs in the workload S are root nodes, i.e., they do not have parents.
- Each AND node's children are OR nodes labeled by the operands of the RPQ operator that labels it.
- Each OR node's children, if it has any, are AND nodes labeled by the lowest-precedence operator in the RPQ associated with it. When there are multiple lowest-precedence operators, specifically / or |, each / constitutes a child, while all the |'s are merged into a child taking all the operands, since /'s execution order affects query time, but |'s does not.

The construction of an RPQ's AODC is a top-down procedure in the descending order of the operators' precedence. To construct the AODC of a workload, we construct each RPQ's AODC, sharing OR nodes whenever possible. The ordering of the workload RPQs does not affect the structure of the resulting AODC. The following example shows the AODC's structure and construction procedure.

Example. Fig. 3 shows the AODC of the workload $\{(a/b/c)^+/(b/c|d^*), a/b/c/d^*, b/c|d\}$, each RPQ having a frequency of 1. Construction begins at a root node labeled by a workload RPQ and proceeds downwards. When there are multiple lowest-precedence operator /'s, e.g., at the OR node labeled $a/b/c$, each / labels one of its children. When the construction finishes from a root node, it starts from the next, until having spanned from all of them. We omit the descendants of $a/b/c/d^*$'s first two children due to limited space. Note how OR nodes can be shared by parent AND nodes, e.g., b/c.

The purpose of the AODC is twofold: the first is to capture the relations between candidate materialized views (i.e., OR nodes) via paths, and the second is to share the computation on common subplans during planning (Sec. 5.3).

5.3 Planning

The AODC incorporates all the possible plans of each workload query, each resulting from a distinct concatenation order. The key to planning is to choose a near-optimal order. Concatenations are similar to relational joins except that they enforce a strict left-to-right semantics. For example, in a relational query, ABC can translate to $(A \bowtie C) \bowtie B$, but the RPQ $a/b/c$ cannot translate to $(a/c)/b$. Hence, the concatenation order is as crucial to an RPQ plan's efficiency as the join order to relational algebra. The planning process chooses the optimal execution tree for each workload query by conducting a depth-first search (DFS) from each respective node, making choices in a bottom-up fashion, as shown in Alg. 1.

Algorithm 1: Query Planning on the AODC

Input: The AODC *aod*, the RPQ workload S, the graph G,
the cost function $cost : V_{aod} \rightarrow \mathbb{R}_0^+$,
the cardinality estimation function $card : V_{aod} \rightarrow \mathbb{N}_0^+$
Output: The annotated AODC *aod*

```
1  forall q ∈ S do
2      planNode(q's node in aod)
3  Function planNode(x):
4      if x has been planned or has no children then
5          return
6      forall y in x.children do
7          planNode(y)
8      if x is an OR node then
9          x.cost ← +∞
10         for i in range(0, x.numChild) do
11             if x.children[i].cost < x.cost then
12                 x.cost ← x.children[i].cost
13                 x.card ← x.children[i].card
14                 x.targetChild ← i
15     else
16         x.cost←estimateCost(x.opType, x.children, G)
17         x.card←estimateCard(x.opType, x.children, G)
18         if x.opType = / then
19             Choose the cheaper direction (→ or ←)
```

Apart from the AODC and the query workload, Alg. 1 also requires inputting a cost and a cardinality function, mapping each AODC node to their estimated costs and cardinalities. We can directly obtain the exact cardinalities of the RPQs of single edge labels, i.e., the leaf nodes in the AODC, and their costs are equal to their cardinalities because answering them is equivalent to scanning all the edges in the data graph labeled by them; the other nodes' costs and cardinalities are initialized as $+\infty$, which `planNode`, the DFS procedure, will refine later.

`planNode` starts by checking if the current AODC node x has been planned, i.e., visited by a prior DFS, and returns if so (Lines 4-5), preventing the recomputation of common subplans. Then it recursively invokes itself on x's children (Lines 6-7). If x is an OR node, it selects the child node with the lowest estimated cost as the target child for execution and sets its cost and cardinality equal to its target child's (Lines 8-14). Otherwise, x is an AND node, so we estimate its cost and cardinality based on its operator type and its children's estimates (Lines 15-17). Particularly, when x is a concatenation node, the direction of its execution (from left to right, $\rightarrow$; or from right to left, $\leftarrow$) can significantly impact the efficiency, so the algorithm estimates their respective costs and chooses the cheaper direction (Lines 18-19). Sec. 7 will introduce the cost and cardinality estimation schemes, `estimateCost` and `estimateCard`.

The output of Alg. 1 is the AODC, each OR node now annotated with its target child, determining the optimal execution trees of each workload query. As a byproduct, Alg. 1 also annotates each AODC node with its estimated cost and cardinality.

Alg. 1 works regardless of whether there are materialized views or not, which only changes the cost function (Sec. 7).

Example. For brevity, we explain the planning process up to the OR node $a/b/c$ in Fig. 3. Planning starts at the leaf nodes, whose costs are equal to their cardinalities as shown in the blue boxes, directly available from the data graph (Fig. 6). We can then estimate the costs and cardinalities of the /'s between single edge labels according to Eqn. 18 and 19 (Sec. 7), which are also the costs and cardinalities of their parents, a/b and b/c. Next, we do estimations on the two child nodes of $a/b/c$, which represent its two concatenation orders. The child on the right, which indicates concatenating b and c first and then a, is selected as the target child for execution due to its lower cost, marked by the red edge in bold. The arrows in the concatenation nodes denote the chosen execution direction.

5.4 Execution

Given an AODC *aod* annotated by Alg. 1, a workload RPQ q in *aod*, and a data graph G, Alg. 2 executes q according to *aod* to get $[\![q]\!]_G$. The execution procedure, `executeNode`, also has a DFS framework. Besides the node to execute, it has two other inputs, *lCand* and *rCand*, which are the candidate source and target data nodes, respectively. These are initialized as empty sets, since there is no constraint on a workload RPQ's source or target. However, Alg. 2 fills them during the execution of concatenation nodes, facilitating sideways information passing [7] between the concatenated subqueries, which reduces the search space (details below).

The base case is OR nodes that have been materialized or single labels, whose results we can directly obtain. If there are candidate source or target data nodes, we apply a filter on the results accordingly (Lines 4-5). The other OR nodes' results need computing. They invoke the execution of their target children, which represent their most efficient execution plan, passing along the candidate sets (Line 6).

The AND nodes' execution differs with their operator types, following Def. 2. The execution of concatenation and Kleene closure nodes are particularly noteworthy. A concatenation node has two execution directions, $\rightarrow$ and $\leftarrow$, executing the left or right child first, respectively. Alg. 1 chooses the cheaper one and annotates the concatenation node accordingly. Taking the $\rightarrow$ direction as an example (Lines 9-14), when executing the left child, we pass in the candidate sources that we currently have. In addition, having executed the left child, its targets can serve as the right child's candidate sources, with the exception of the left child's result set having an ϵ, where the right child's sources should be unconstrained. The case is similar for the $\leftarrow$ direction (Lines 15-20). The Kleene closure nodes computes the fix-point of its child node's result set (Lines 34-40). In practice, we append the newly produced results to the end of *childRes* in each iteration, so we don't need to explicitly implement the set minus in Line 35 or the `unionRes` call in Line 40.

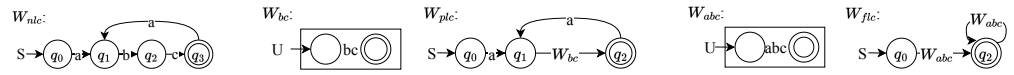
The lazy treatment of ϵ. As described in Sec. 5.2, an ϵ produced by the ? or $*$ operator will lead to $\{\langle v, v\rangle \mid v \in V\}$ in its results, which contains $|V|$ tuples. Instead of immediately materializing this set when encountering an ϵ, we mark the respective nodes' result sets with ϵ to save memory and time (Lines 29 and 38). When a marked result set joins with another on the left (right), `joinRes` implements a right-(left-)outer join. If a workload query's result set is marked, it will finally emit $\{\langle v, v\rangle \mid v \in V\}$.

Example. Like the planning process, the execution of the AODC in Fig. 3 starts at the leaf nodes and proceeds upwards. Taking b/c as an example, if it has not been materialized, we follow the $\rightarrow$ direction annotating its child concatenation node, and try matching the target nodes in $[\![b]\!]_G$ to source nodes in $[\![c]\!]_G$, getting two matches, which form $[\![b/c]\!]_G$. The results are shown in the green tables in Fig. 3. Afterwards, we join $[\![b/c]\!]_G$ with $[\![a]\!]_G$ to get $[\![a/b/c]\!]_G$ according to the concatenation order dictated by the chosen target child. Note that the targets of $[\![a/b/c]\!]_G$ serving as d^*'s candidate sources when executing $a/b/c/d^*$ greatly reduces the search space, since most of

Algorithm 2: Executing a query on the AODC

```
Input: The AODC aod, the workload RPQ q, the graph G
Output: [[q]]_G
executeNode(q's node in aod, ∅, ∅)
Function executeNode(x, lCand, rCand):
    if x is an OR node then
        if x is materialized or a single label then
            return filter(x.res, lCand, rCand)
        return executeNode(x.targetChild, lCand, rCand)
    else
        if x.opType = / then
            if x.direction = → then
                lRes ← executeNode(x.children[0], lCand, ∅)
                if lRes has ϵ then
                    rRes ← executeNode(x.children[1], ∅, rCand)
                else
                    rRes ← executeNode(x.children[1], lRes.t, rCand)
            else
                rRes ← executeNode(x.children[1], ∅, rCand)
                if rRes has ϵ then
                    lRes ← executeNode(x.children[0], lCand, ∅)
                else
                    lRes ← executeNode(x.children[0], lCand, rRes.s)
            return joinRes(lRes, rRes)
        else if x.opType = | then
            curRes ← ∅
            forall y in x.children do
                curRes ← unionRes(curRes, executeNode(y, lCand, rCand))
            return curRes
        else if x.opType = ? then
            curRes ← executeNode(x.children[0], lCand, rCand)
            Mark curRes with ϵ
            return curRes
        else if x.opType = + or x.opType = ∗ then
            childRes ← executeNode(x.children[0], lCand, rCand)
            deltaRes ← childRes, curRes ← childRes
            while true do
                deltaRes ← joinRes(deltaRes, childRes) - curRes
                if deltaRes = ∅ then
                    if x.opType = ∗ then
                        Mark curRes with ϵ
                    return curRes
                curRes ← unionRes(curRes, deltaRes)
```

the d edges in the data graph (Fig. 6) do not follow an $a/b/c$ path, thus excluded from the search. On the other hand, if b/c has been materialized, we can directly obtain its results.



(a) No LC (b) Partial LC (c) Full LC

Fig. 4. Waveplans with different LC strategies.

Executing queries outside of the workload. Our method can support planning and executing queries that are not in the original workload. Specifically, given such a query, we first add it to the AODC, then run Alg. 1 and Alg. 2 on it. If this query shares a subplan with a workload query, Alg. 1 will skip the subplan's planning, preventing recomputation.

5.5 Discussion: Comparison with Waveplans

The state-of-the-art RPQ plan representation, Waveplans [27], has a plan space that properly subsumes those of finite automata and relational algebra extended with the fix-point operator α (α-RA). The plans that Waveplan can express but α-RA cannot be "no loop caching" and "partial loop caching" plans for Kleene closures. In contrast, they call the fix-point operator in α-RA "full loop caching". Fig. 4 shows the Waveplans for $(a/b/c)^+$ using no/partial/full loop caching (LC), respectively. The fundamental difference between the different LC strategies is how much of the cycle representing the Kleene closure in the automaton is computed in a separate automaton. No LC does not have a separate automaton, partial LC has a separate automaton for part of the loop, and full LC has a separate automaton for the whole loop.

The AODC is equivalent to multiple α-RA plans sharing their common sub-plans, implementing Kleene closures as full LC. Therefore, the plan space of AODC is the same as that of α-RA. Though it can be extended to support no LC and partial LC plans by annotating the Kleene closure nodes accordingly, we find in our preliminary experiments that no LC and partial LC plans are hardly ever faster than α-RA plans, which is consistent with the findings in Sec. 5.2 of [27]. We thus exclude these plans in this work.

We also note that it is possible but difficult to design a multi-RPQ plan representation based on Waveplans, since the graph structure of automata complicates the merging of alternative plans.

6 MATERIALIZED VIEW SELECTION

As explained in Sec. 5.1, a straightforward greedy solution to the hard MVS for RPQ problem can lead to many redundant materialized views, not only wasting memory but also obstructing other useful views from materialization. We thus introduce the AODC for redundancy detection and removal. Sec. 6.1 gives our MVS algorithm (Alg. 4) and explains how it uses the AODC to detect redundant views. Since whether a view is redundant depends on the current AODC, Alg. 4 requires an algorithm that updates the AODC with changes in the view materialization status, which we introduce in Sec. 6.2 (Alg. 5).

6.1 View Selection Algorithm

Heuristic. We adopt the frequency of candidate views in the workload as the ordering heuristic in the greedy selection scheme. We have tried other heuristics, such as the frequency multiplied by the cost or the difference between the cost and the cardinality of the candidate view, since these are upper bounds on how much cost it can save. However, their experimental effects are all slightly less significant than the frequency heuristics, probably due to the looseness of the upper bounds.

Algorithm 3: AODC node's usage count maintenance

```
   Input: The AODC node x, the delta usage count δ
1  Function propagateUseCnt(x, δ):
2      useCnt[x] ← useCnt[x] + δ
3      if x has no children then
4          return
5      if x is an OR node then
           /* x.targetChild is x's chosen child for execution (Alg. 1)   */
6          propagateUseCnt(x.targetChild, δ)
7      else
8          for y in x.children do
9              propagateUseCnt(y, δ)
```

Basic idea of redundancy detection. The key idea is: a materialized view is redundant if and only if the AODC node denoting it is not used in any workload RPQ's optimal execution tree chosen by Alg. 1 We define the number of optimal execution trees that an AODC node appear in as its *usage count*. A view is redundant if and only if its usage count is 0. We maintain each node's usage count for redundancy detection with Alg. 3. Alg. 3 has the same DFS framework as Alg. 2, but it only traverses the target child when encountering an OR node to track whether a node is in the optimal execution trees. We call Alg. 3 at two points in the workflow:

- After Alg. 1: for each RPQ q in the workload, initialize the usage count of its node as 0, then call `propagateUseCnt`(q's node, 1);
- During Alg. 4: when the AODC updates due to newly materialized views.

MVS algorithm. Alg. 4 is our greedy MVS algorithm using the AODC to detect and remove redundant views. The memory budget b is in terms of the total cardinality, i.e., result node pairs. We first sort the candidate views in the descending order of frequency (Line 1) and initialize the amount of memory used as 0 and the set of materialized views as empty (Lines 2-3). Then we scan each candidate view in the sorted order, skipping a candidate view if it is redundant or will cause exceeding the memory budget (Lines 5-6). A qualified candidate view triggers the AODC update procedure, `replanWithMat` (Alg. 5), which returns the candidate view's benefit (i.e., the workload query cost reduced by its materialization) and the changes in the AODC, including the changes in the nodes' costs and target children (Line 7). If the candidate view does not reduce the workload query cost and has non-zero estimated cardinality, we skip it without applying Δ to *aod* (Lines 8-9). We keep the views with zero benefit and zero estimated cardinality because they are not expected to take up much memory and are thus harmless to materialize. The candidate views that pass the three tests above will be materialized, so we apply Δ to *aod* (Line 10) and increment the used amount of memory accordingly (Line 11).

The key to view redundancy detection using the AODC is maintaining the usage counts in `applyChanges`. Changes in an AODC node's usage count reflect it leaving or joining optimal execution trees in the following occasions:

- An AND node that was originally an OR node's target child but gets replaced due to a view's materialization causes itself and all its descendants in its optimal execution tree to *leave* the trees that its parent OR node is in (Line 20);
- An AND node that newly becomes the target child of an OR node causes itself and all its descendants in its optimal execution tree to *join* the trees that its parent OR node is in (Line 21);

Algorithm 4: MVS for RPQs

```
   Input: The AODC aod, the memory budget b
   Output: The set of materialized views V
1  rankedViews ← sort(candViews, freq, desc)
2  usedMem ← 0
3  V ← ∅
4  forall x in rankedViews do
5      if useCnt[x] = 0 or usedMem + x.card > b then
6          continue
7      benefit, Δ ← replanWithMat(aod, x)
8      if benefit = 0 and x.card > 0 then
9          continue
10     applyChanges(aod, x, Δ)
11     usedMem ← usedMem + x.card
12     V ← V ∪ {x}
13     forall y ∈ V do
14         if useCnt[y] = 0 then
15             usedMem ← usedMem - y.card
16             V ← V − {y}
17 Function applyChanges(aod, x, Δ):
18     Update the costs and cardinalities according to Δ
19     forall y whose target child changed in Δ do
20         propagateUseCnt(y.oldTargetChild, -useCnt[y])
21         propagateUseCnt(y.targetChild, useCnt[y])
22     tmpUseCnt ← useCnt[x]
23     propagateUseCnt(x, -tmpUseCnt)
24     useCnt[x] ← tmpUseCnt
```

- A newly materialized node causes all its descendants in its optimal execution tree to *leave* the tree, since they will no longer be executed to get the materialized node's results. Note that the materialized node's usage count does not change (Lines 22-24). Intuitively, its descendants are detached from it.

Therefore, `applyChanges` may reduce some of the already selected views' usage counts to 0, meaning that they are rendered redundant by newly selected views. They are thus removed from the set and the memory budget is restored to incorporate more useful views (Lines 13-16). Since these views have zero usage count, indicating that no workload query uses them, removing them will not change the AODC, so there is no need to call `replanWithMat`.

Note that Alg. 4 will never wrongly remove a view that is non-redundant in the final state, because once a view becomes redundant during selection, it will always be redundant. The intuition is: even if the view B that originally makes the view A redundant is removed from the set, it would be because a newly materialized view C has made B redundant; therefore, C also makes A redundant.

Remarkably, Alg. 4 also gives us the view-based query plans with the lowest estimated cost, integrating the two workflows.

Example. Tab. 3 shows a possible MVS process of the AODC in Fig. 3 with the memory budget $b = 12$; there are multiple possibilities because Alg. 4 breaks ties arbitrarily when candidate views have the same frequency. V is the set of selected views after examining x; usedMem is the sum

Table 3. Example MVS process.

	x (Candidate view)	V (Selected view set)	usedMem
Step 1	b/c	$\{b/c\}$	2
Step 2	d^*	$\{b/c\}$	2
Step 3	$a/b/c$	$\{b/c, a/b/c\}$	10
Step 4	a/b	$\{b/c, a/b/c\}$	10

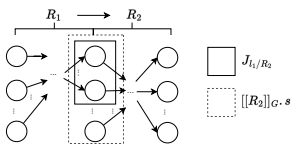


Fig. 5. The effect of sideways information passing.

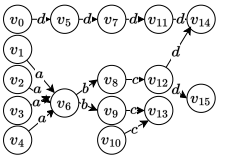


Fig. 6. Example data graph.

of the views' cardinalities in V. Alg. 4 selects the three views with positive benefits out of the five candidates examined. Even though the selected b/c and $a/b/c$ hold a string subsumption relation, each is non-redundant, since materializing $a/b/c$ reduces the cost of executing $(a/b/c)^+$ and $a/b/c/d^*$ further than solely materializing b/c, and materializing b/c speeds up the execution of the workload query $b/c|d$ and the subquery $(b/c|d^*)$. d^* is not selected because its large cardinality will cause exceeding the budget; a/b is not selected because $a/b/c$ renders it redundant. Though not shown in Tab. 3, all the other candidate views are examined after a/b, but none is selected because of either exceeding the budget or redundancy.

Materializing the selected views. After MVS, we need to materialize the selected views. We can directly invoke Alg. 2 to do so, while being careful to flag a view as materialized after finishing its execution. Since a materialized view can be used during other views' materialization, we materialized the set of selected views in descending topological order of the AODC, i.e., materializing the views close to the leafs first so that their ancestors can use them.

6.2 Incremental View-Based Planning

In Alg. 4, each time a candidate view is newly selected for materialization, we update the AODC by calling the `replanWithMat` procedure. The fundamental reason why a view's materialization affects the query plan is that an AODC node's cost turns into its cardinality when materialized, since instead of computing its result by matching it onto the data graph, we can directly read its precomputed results. Thus, a naive implementation of `replanWithMat` would be to call Alg. 1 again with a cost function accordingly updated. However, it is more efficient to incrementally update the plan, as shown in Alg. 5.

Incremental update of the AODC is possible because the change in a node's cost will only affect its ancestors'. Alg. 5 propagates the cost change by a reverse DFS from the materialized node x. When the reverse DFS visits an ancestor node, it recomputes the ancestor node's cost based on the updated costs of its children using the cost estimation scheme in Sec. 7. When a visited ancestor node represents a workload query, we accumulate its cost reduction into the benefit, taking its frequency into account (Lines 8-9). Though `replanNode`'s pseudocode directly updates the AODC for convenience, we actually save the changes in Δ and refrain from applying them before ensuring that we will indeed materialize x (Alg. 4).

Notably, the reverse DFS in Alg. 5 does not necessarily traverse all the ancestors of x, but stops traversing from an ancestor when its cost does not change (Lines 6-7).

6.3 Discussion: MVS for RPQ in Graph DBMS

MVS for RPQ clearly benefits applications that directly evaluate an RPQ workload. However, with the wide usage of graph DBMS (e.g., as the backend of SPARQL query endpoints), it is worth discussing how to apply the proposed MVS for RPQ techniques in the context of graph DBMS. As introduced in Sec. 1, graph DBMS query languages such as SPARQL, ISO/GQL, and openCypher at

Algorithm 5: Incremental View-Based Planning

```
Input: The AODC aod, the node to materialize x
Output: The benefit of x, The changes in the AODC Δ
Function replanWithMat(aod, x):
    benefit ← 0
    replanNode(aod, x, x.card, benefit, Δ)
    return benefit, Δ
Function replanNode(aod, x, newCost, benefit, Δ):
    if newCost >= x.cost then
        return
    if x's query is in the workload then
        benefit ← benefit + (cost[x] - newCost) × freq[x]
    cost[x] ← newCost
    if x is an OR node then
        forall y in x.parents do
            replanNode(aod, y, newCost, benefit)
    else
        forall y in x.parents do
            newCost ← estimateCost(y.opType, y.children, G)
            if x has the lowest cost in y.children then
                y.targetChild ← x
            if y.opType = / then
                Choose the cheaper direction (→ or ←)
```

least partially support RPQ, but there are often other constructs in the query, making it non-trivial to apply our MVS for RPQ techniques.

On one hand, the processing of a graph DBMS query workload can benefit from our method as long as they contain RPQs, since we can extract their RPQ components as the input to our method, which returns a set of materialized views for RPQ. During query execution, if the sources or targets of the RPQ's result are constrained by other query constructs, a lookup in the respective materialized view using the constrained source or target nodes as keys finds the results.

On the other hand, the above approach will not lead to the optimal efficiency, since it does not optimize the execution order of possibly multiple RPQs and the other query constructs. To our knowledge, the only work that considers optimizing this order is [4], which assumes fixed minimal DFA plans for RPQs. We believe optimizing the execution order based on query planning and MVS for RPQ is an important direction for future work.

7 COST & CARDINALITY ESTIMATION

Both the query planning (Alg. 1) and the materialized view selection (Alg. 4) algorithms depend on the cost and cardinality estimates of the workload RPQs and their subqueries. We devise our own cost and cardinality estimation schemes, since no existing method meets our needs to our knowledge. Sec. 7.1 will introduce the recently proposed cardinality estimation method in [7] for RPQs with concatenations only and the reasons why it is not suited to our setting. Sec. 7.2 will give our method that improves on theirs.

Assumption on the storage layout. Our method works as long as iterating over a node v's out- or in-neighbors connected by edges labeled l takes time linear to the number of such neighbors,

supported by most native graph databases with indexes for each node's neighborhood, regardless of the graph data model [6].

7.1 Existing Approach

Sec. 8.3.1 of [7] describes a cardinality estimation method for RPQs with concatenations only. Since our approach stems from theirs, we introduce the derivation of their estimations in detail. Their method starts from concatenating two edge labels l_1/l_2, observing that the following equation calculates its exact cardinality:

$$|[\![l_1/l_2]\!]_G| = \sum_{j \in J_{l_1/l_2}} |\sigma_{t=j}([\![l_1]\!]_G)| \cdot |\sigma_{s=j}([\![l_2]\!]_G)| \tag{9}$$

where the join set J_{l_1/l_2} is the set of data nodes that are both targets of l_1 and sources of l_2 in G, i.e., $J_{l_1/l_2} = [\![l_1]\!]_G.t \cap [\![l_2]\!]_G.s$. However, it is infeasible to compute and store J for each label pair on large graphs, since it takes $O(|V| \cdot |\Sigma|^2)$ time and memory. To solve this problem, [7] introduces the uniformity assumption, i.e., all nodes in the join set have the same number of tuples associated with them in both $[\![l_1]\!]_G$ and $[\![l_2]\!]_G$. They thus simplify Eqn. 9 as Eqn. 10:

$$|[\![l_1/l_2]\!]_G| = |J_{l_1/l_2}| \cdot \frac{|[\![l_1]\!]_G|}{|[\![l_1]\!]_G.t|} \cdot \frac{|[\![l_2]\!]_G|}{|[\![l_2]\!]_G.s|} \tag{10}$$

where $|[\![l_1]\!]_G|$, $|[\![l_1]\!]_G.t|$, $|[\![l_2]\!]_G|$, and $|[\![l_2]\!]_G.s|$ are all directly available from the graph, so they only need to estimate $|J_{l_1/l_2}|$.

They then extendEqn. 10 to the more general case of estimating $|[\![R/l_2]\!]_G|$:

$$|[\![R/l_2]\!]_G| = |J_{R/l_2}| \cdot \frac{|[\![R]\!]_G|}{|[\![R]\!]_G.t|} \cdot \frac{|[\![l_2]\!]_G|}{|[\![l_2]\!]_G.s|} \tag{11}$$

where $|[\![R]\!]_G|$ and $|[\![R]\!]_G.t|$ are from the previous estimation step.

To estimate $|J_{R/l_2}|$, they rewrite R in the form of $R'l_1$, where l_1 is R's ending label. Considering a data node v in $[\![l_1]\!]_G.t$, the probability that it is in J_{R/l_2} is:

$$\Pr[v \in J_{R/l_2}] = \Pr[v \in [\![R]\!]_G.t \cap J_{l_1/l_2}] \tag{12}$$

Assuming $v \in [\![R]\!]_G.t$ and $v \in J_{l_1/l_2}$ are independent, they derive:

$$\Pr[v \in J_{R/l_2}] = \frac{|[\![R]\!]_G.t|}{|[\![l_1]\!]_G.t|} \cdot \frac{|J_{l_1/l_2}|}{|[\![l_1]\!]_G.t|} \tag{13}$$

$|J_{R/l_2}|$ follows a binomial distribution with $p = \Pr[v \in J_{R/l_2}]$ and $n = |[\![l_1]\!]_G.t|$, so they can estimate it as its expectation as follows:

$$\mathbb{E}[|J_{R/l_2}|] = n \cdot p = \frac{|[\![R]\!]_G.t| \cdot |J_{l_1/l_2}|}{|[\![l_1]\!]_G.t|} \tag{14}$$

Substituting $|J_{R/l_2}|$ by $\mathbb{E}[|J_{R/l_2}|]$ in Eqn. 10, they get the final estimation of R/l_2's cardinality:

$$|[\![R/l_2]\!]_G| = \frac{|J_{l_1/l_2}| \cdot |[\![R]\!]_G| \cdot |[\![l_2]\!]_G|}{|[\![l_1]\!]_G.t| \cdot |[\![l_2]\!]_G.s|} \tag{15}$$

where $|[\![l_2]\!]_G|$, $|[\![l_1]\!]_G.t|$, and $|[\![l_2]\!]_G.s|$ are directly available from the data graph and $|[\![R]\!]_G|$ is available from the previous estimation step. [7] uses a synopsis to record $|J_{l_1/l_2}|$ for each label pair, which requires $O(|E| \cdot |\Sigma|)$ computation time and $O(|\Sigma|^2)$ memory.

Besides, [7] also estimates R/l_2's distinct number of source and target data nodes assuming uniformity:

$$|[\![R/l_2]\!]_G.s| = |[\![R]\!]_G.s| \cdot \frac{|J_{l_1/l_2}|}{|[\![l_1]\!]_G.t|} \tag{16}$$

$$|[\![R/l_2]\!]_G.t| = |[\![R]\!]_G.t| \cdot \frac{|[\![l_1/l_2]\!]_G.t|}{|[\![l_1]\!]_G.t|} \tag{17}$$

where $|[\![l_1/l_2]\!]_G.t|$ is also collated as part of the synopsis.

If R has multiple ending labels as the result of |, $*$, or ?, it is necessary to substitute l_1 by each ending label in Eqn. 15, Eqn. 16, or Eqn. 17 and sum them to get $|[\![R/l_2]\!]_G|$, $|[\![R/l_2]\!]_G.s|$, or $|[\![R/l_2]\!]_G.t|$.

Remarks. [7]'s cardinality estimation method does not suit our setting due to two reasons. Firstly, the estimation scheme of extending a single label at a time does not suit the AODC, whose concatenation operator concatenates two arbitrary RPQs. Secondly, the synopsis, which consists of the exact join set sizes of all the edge label pairs, is too expensive to construct on large graphs with many edge labels. For example, the synopsis of the Wikidata graph [24] used in our experiments takes over two hours to construct. Our approach addresses these problems.

We are also aware of a recent work on RPQ cost estimation [18], but it only applies to finite automata (FA) plans, not AODC plans.

7.2 Our Approach

We devise cost and cardinality estimation schemes suited to our AODC-based query planning procedure by extending the existing approach (Sec. 7.1). Specifically, we extend the existing cardinality estimation method to the general case of concatenating two arbitrary RPQs and replace the time-consuming synopsis construction by Monte Carlo sampling. Besides, we propose a cardinality estimation method for Kleene closures, which has not been studied in the literature to our knowledge.

7.2.1 Concatenation. We extend [7]'s cardinality estimation method to handle concatenating arbitrary RPQs. Specifically, we substitute the label l_2 by the arbitrary RPQ R_2 in Eqn. 15, extending the uniformity and independence assumptions accordingly:

$$|[\![R_1/R_2]\!]_G| = \frac{|J_{l_1/R_2}| \cdot |[\![R_1]\!]_G| \cdot |[\![R_2]\!]_G|}{|[\![l_1]\!]_G.t| \cdot |[\![R_2]\!]_G.s|} \tag{18}$$

where the newly emerged $|[\![R_2]\!]_G|$ and $|[\![R_2]\!]_G.s|$ are available as the estimations of the concatenation node's children. On the other hand, $|J_{l_1/R_2}|$, if stored as part of a synopsis like $|J_{l_1/l_2}|$, will lead to a synopsis that's larger and more expensive to compute. Fortunately, we can estimate it by Monte Carlo sampling (Sec. 7.2.5).

The cost of the concatenation node consists of its children's evaluation cost and the cost of joining their results. Notably, due to the sideways information passing [7] optimization on Alg. 2, if the child executed first does not have an ϵ, the other child's evaluation cost will decrease because of the constraints on its source or target data nodes. Fig. 5 illustrates this phenomenon, taking the $\rightarrow$ direction as an example, where R_2's sources are constrained.

Therefore, we estimate a concatenation node's cost as Eqn. 19:

$$cost(R_1/R_2) = \min\{cost(R_1) + \frac{|J_{l_1/R_2}|}{|[\![R_2]\!]_G.s|} cost(R_2), cost(R_2) + \frac{|J_{l_1/R_2}|}{|[\![R_1]\!]_G.t|} cost(R_1)\} + |[\![R_1]\!]_G| + |[\![R_2]\!]_G| \tag{19}$$

where $|[\![R_1]\!]_G| + |[\![R_2]\!]_G|$ is the estimated join cost.

We can also estimate the distinct number of source and target nodes of R_1/R_2, substituting l_2 by R_2, which results in $|[\![l_1/R_2]\!]_G.t|$ in the numerator of Eqn. 17. Since it is not available unless we execute l_1/R_2, we expand it as follows by the uniformity assumption:

$$|[\![l_1/R_2]\!]_G.t| = |J_{l_1/R_2}| \cdot \frac{|[\![R_2]\!]_G.t|}{|[\![R_2]\!]_G.s|} \tag{20}$$

Combining Eqn. 16, Eqn. 17, and Eqn. 20, we estimate the distinct number of source and target nodes of R_1/R_2 as follows:

$$|[\![R_1/R_2]\!]_G.s| = |[\![R_1]\!]_G.s| \cdot \frac{|J_{l_1/R_2}|}{|[\![l_1]\!]_G.t|} \tag{21}$$

$$|[\![R_1/R_2]\!]_G.t| = |[\![R_1]\!]_G.t| \cdot \frac{|J_{l_1/R_2}|}{|[\![l_1]\!]_G.t|} \cdot \frac{|[\![R_2]\!]_G.t|}{|[\![R_2]\!]_G.s|} \tag{22}$$

7.2.2 Kleene closures. Since we lazily handle ϵ (Sec. 5.4), a Kleene closure's result size in memory is the same whether the operator is $*$ or $+$. Hence, without loss of generality, we discuss the cardinality estimation of $+$ in the following.

The following Eqn. 23 represents the exact cardinality of R^+:

$$|[\![R^+]\!]_G| = |[\![R]\!]_G| + \sum_{i=2}^{\infty} |[\![R^i]\!]_G - \bigcup_{j=1}^{i-1} [\![R^j]\!]_G| \tag{23}$$

Intuitively, Eqn. 23 gathers the newly produced result tuples step by step, subtracting all the previous tuples, $\bigcup_{j=1}^{i-1}[\![R^j]\!]_G|$, from the current concatenation step's result $[\![R^i]\!]_G$, like the fix-point execution in Lines 34-40, Alg. 2.

We can estimate $||[\![R^i]\!]_G||$ $(i > 1)$ using the cardinality estimation method of concatenation (Eqn. 18), since $R^i = R^{i-1}/R$:

$$|[\![R^i]\!]_G| = \frac{|J_{l_1/R}| \cdot |[\![R]\!]_G|}{|[\![l_1]\!]_G.t| \cdot |[\![R]\!]_G.s|} \cdot |[\![R^{i-1}]\!]_G| \tag{24}$$

where l_1 is R's ending label. Interestingly, we can view Eqn. 24 as a recursive formula that computes $|[\![R^i]\!]_G|$ from $|[\![R^{i-1}]\!]_G|$ and the fraction on its right hand side as a constant coefficient, since all the estimations related to R are available at the time of estimating $|[\![R^+]\!]_G|$. We denote this constant coefficient by c. Thus, recursively expanding Eqn. 24 will give us:

$$|[\![R^i]\!]_G| = c^{i-1}|[\![R]\!]_G| \tag{25}$$

Though we can now estimate $|[\![R^i]\!]_G|$ for any i, it is still difficult to estimate $|[\![R^i]\!]_G - \bigcup_{j=1}^{i-1}[\![R^j]\!]_G|$. We leave it as future work and make the following relaxation, which is always an overestimate:

$$|[\![R^i]\!]_G - \bigcup_{j=1}^{i-1}[\![R^j]\!]_G| \approx |[\![R^i]\!]_G| \tag{26}$$

Hence combining Eqn. 23, Eqn. 25, and the relaxation gives us the final cardinality estimation formula for Kleene closures:

$$|[\![R^+]\!]_G| = \sum_{i=0}^{D-1} c^i |[\![R]\!]_G| = \frac{1-c^D}{1-c}|[\![R]\!]_G| \tag{27}$$

where D is the maximum number of useful concatenation steps, i.e., the number of iterations that the fix-point procedure terminates at. That is, if $[\![R^{i-1}]\!]_G$ still produces new tuples but $[\![R^i]\!]_G$ does not, $D = i - 1$. We cannot know d exactly without executing R^+. However, due to Eqn. 25, when $c < 1$, we can bound d by $c^{D-1}|[\![R]\!]_G| >= \epsilon$ and $c^D|[\![R]\!]_G| < \epsilon$, where ϵ is a threshold parameter, meaning that nearly no new result tuples will emerge beyond D steps. On the other hand, when $c >= 1$, we fix $D = 6$ according to the small world assumption [17]. Note that the number of new result tuples will not actually increase indefinitely with $c >= 1$; Eqn. 27 exhibits this phenomenon because of the relaxation (Eqn. 26).

Regarding the distinct number of source data nodes, we observe that $\forall i >= 1$, $[\![R^i]\!]_G.s \in [\![R]\!]_G.s$, since any R^i's evaluation starts from R. The case is similar for target nodes. Therefore, we have

$$[\![R^+]\!]_G.s = [\![R]\!]_G.s \tag{28}$$

$$[\![R^+]\!]_G.t = [\![R]\!]_G.t \tag{29}$$

and the equality of their cardinalities follows.

Lastly, we discuss the cost of evaluating a Kleene closure. Extending on Eqn. 19, each concatenation step incurs $\frac{|J_{R_i/R}|}{|[\![R]\!]_G.s|} \cdot cost(R)$, where $|J_{R_i/R}| = \frac{|[\![R^i]\!]_G.t| \cdot |J_{l_1/R}|}{|[\![l]\!]_G.t|}$ and l_1 is R's ending label. Since we have inferred $\forall i, [\![R^i]\!]_G.t \subseteq [\![R]\!]_G.t$ during the derivation of Eqn. 28 and Eqn. 29, we uniformly approximate $|[\![R^i]\!]_G.t|$ by $|[\![R]\!]_G.t|$, which gives us the following cost estimation formula:

$$cost(R^+) = (1 + (D-1) \cdot \frac{|[\![R]\!]_G.t| \cdot |J_{l_1/R}|}{|[\![R]\!]_G.s| \cdot |[\![l]\!]_G.t|}) \cdot cost(R) + (D - 1 + \frac{1-c^D}{1-c})|[\![R]\!]_G| \tag{30}$$

where $(D-1+\frac{1-c^D}{1-c})|[\![R]\!]_G|$ is the estimated join cost.

7.2.3 Alternation. We estimate an alternation node's cardinality and distinct number of source and target data nodes as the sum of its children's respective estimates. Note that we apply a similar relaxation as Eqn. 26, discounting the set overlaps:

$$|[\![R_1|R_2|\cdots|R_k]\!]_G| = |\bigcup_{i=1}^{k}[\![R_i]\!]_G| \approx \sum_{i=1}^{k}|[\![R_i]\!]_G| \tag{31}$$

An alternation node's cost is the sum of its children's evaluation costs and the set union cost, which we estimate by the sum of the children's cardinalities:

$$cost(R_1|R_2|\cdots|R_k) = \sum_{i=1}^{k}(cost(R_i)+|[\![R_i]\!]_G|) \tag{32}$$

7.2.4 Zero-or-one. Due to the lazy treatment of ϵ (Sec. 5.4), the estimates of a zero-or-one node's cost, cardinality, and distinct number of source and target data nodes are equal to its child's.

7.2.5 Monte Carlo Sampling. We use Monte Carlo sampling for estimating the sizes of join sets, $|J_{l/R}|$, where l is a single label and R is an arbitrary RPQ. Recall that [7] uses a synopsis of the graph to record the exact join set sizes of label pairs. We use Monte Carlo sampling instead of synposes to reduce the computation cost and to support the case where R is an arbitrary RPQ.

The basis of estimating $|J_{l/R}|$ by Monte Carlo sampling is: considering a data node $v \in [\![l]\!]_G.t$, if we can estimate the probability that it is in $J_{l/R}$, we can estimate the join set size by $\mathbb{E}[|J_{l/R}|] = \Pr[v \in J_{l/R}] \cdot |[\![l]\!]_G.t|$, similar to Eqn. 14. Suppose we sample s nodes independently and uniformly at random from $[\![l]\!]_G.t$ and $p \cdot s$ of them are in $[\![R]\!]_G.s$, we can estimate the join set size with:

$$|J_{l/R}| = p \cdot |[\![l]\!]_G.t| \tag{33}$$

To get p in Eqn. 33, we need to check if each sampled node is in $[\![R]\!]_G.s$. For this purpose, we compile the minimum DFA for R and conduct a DFA-guided DFS from the sampled node v in the data graph, which immediately returns true when it finds a result tuple, preventing the extra cost of computing the full $[\![R]\!]_G$.

Example. Fig. 6 shows the example data graph serving as the basis of the planning, execution, and MVS of the AODC in Fig. 3. Here, we use it to illustrate the cost and cardinality estimation of concatenation and Kleene closures, taking b/c and d^* as examples. (Due to lazily treating ϵ (Sec. 5.4), it suffices to do estimations for d^+.) For simplicity, we assume the Monte Carlo sampling returns the exact join set sizes and $D=6$. Note that b/c and d^+'s estimated cardinalities are close to the real values, which are 2 and 12, respectively. b/c's execution direction is chosen as $\longrightarrow$ due to its lower cost.

$$|[\![b/c]\!]_G| = \frac{|J_{b/c}|\cdot|[\![b]\!]_G|\cdot|[\![c]\!]_G|}{|[\![b]\!]_G.t|\cdot|[\![c]\!]_G.s|} = \frac{2\times2\times3}{2\times3} = 2 \tag{34}$$

$$cost(b/c) = \min\{cost(b)+\frac{|J_{b/c}|}{|[\![c]\!]_G.s|}cost(c), cost(c)+\frac{|J_{b/c}|}{|[\![b]\!]_G.t|}cost(b)\}+|[\![b]\!]_G|+|[\![c]\!]_G| = 9$$

$$|[\![d^+]\!]_G| = \frac{1-c^D}{1-c}|[\![d]\!]_G| \approx 11\ (c = \frac{|J_{d/d}|\cdot|[\![d]\!]_G|}{|[\![d]\!]_G.t|\cdot|[\![d]\!]_G.s|} = \frac{3\times6}{6\times6} = 0.5) \tag{35}$$

$$cost(d^+) = (1+(D-1)\cdot\frac{|[\![d]\!]_G.t|\cdot|J_{d/d}|}{|[\![d]\!]_G.s|\cdot|[\![d]\!]_G.t|})\cdot cost(d)+(D-1+\frac{1-c^D}{1-c})|[\![d]\!]_G| = 57 \tag{36}$$

8 EXPERIMENTAL EVALUATION

8.1 Setting

8.1.1 Environment. We conduct all our experiments on a machine with an Intel Xeon 2.1GHz CPU and 1TB RAM. We report a run as out of memory when the memory usage exceed 256GB. All

the evaluated algorithms are implemented in C++[2]. All the experiments run sequentially unless explicitly specified otherwise.

8.1.2 Dataset & Workload. We use the Wikidata graph [24] as the dataset and the RPQs extracted from the Wikidata Query Logs [14] as the workload. The Wikidata graph has $|V| = 348,945,080$ nodes, $|E| = 958,844,164$ edges, and $|\Sigma| = 5,419$ edge labels. During the experiments, we load the graph into memory with a compressed sparse row (CSR) structure for each edge label, taking up 61GB memory. We also store the results of each materialized view as a CSR in main memory. The Wikidata Query Logs consist of SPARQL queries, some of which contains property path clauses, i.e., RPQs in SPARQL syntax. Following the methodology in [5], we extract the RPQs from the code-500 (timeout) sections of [14], which include the SPARQL queries taking over 60 seconds to answer by the Wikidata SPARQL service endpoint, disregarding constant source and target nodes as dictated by Def. 1. We filter out the RPQs with non-existent edge labels. We also filter out the RPQs subsuming the pattern $a/a-$ or $a-/a$, since they return all the wedges in the graph with a-edges and are unlikely to optimize unless by factorization, which is beyond the scope of this paper. After filtering, we get 1,930 queries, 442 of which are distinct. The most frequent query patterns and their respective percentages are shown in Tab. 5.

8.2 Effectiveness of Query Planning with AODC

In this experiment, we compare the efficiency of AODC query plans generated by Alg. 1 and minimal DFA query plans without materialized views in terms of the workload's total query time.

The total query time using the AODC plan without any materialized views is 30187.6 s, while that of using the minimal DFAs is 68552.4 s. The speedup is 2.27×, proving the AODC-based query planning effective. Note that the AODC-based query planning (Alg. 1) takes 13.9 s, which is negligible compared with the query time.

To further examine the contributing factors to the speedup, we plot the speedup distribution of AODC plans without materialized views compared with minimal DFA plans in Fig. 7. Since the workload contains duplicate RPQs, we plot both the original and the deduplicated distributions. We observe that the majority of queries achieve over 2× speedup by using AODC plans. Interestingly, quite a few queries achieve over 10× or even 100× speedup, which are mainly queries with few final results but many intermediate results if executed according to the minimal DFA. These queries benefit much from AODC-based query planning, which detects a favorable ordering of operators leading to fewer intermediate results.

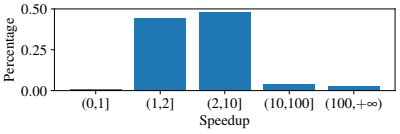
On the other hand, some queries run slower with AODC plans than minimal DFA plans, the lowest speedup being 0.58×. These are queries with Kleene closures that are processed faster by the no LC or partial LC plans than the full LC plans, while the AODC currently supports full LC plans only. Since such queries are scarce, they do not impact the workload's total query time significantly.

8.3 Proof of Concept: MVS for RPQ

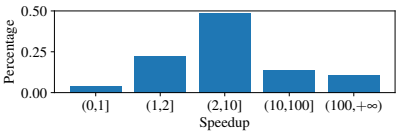
The purpose of this proof of concept experiment is to show the necessity of designing a MVS for RPQ algorithm that considers subqueries and the relations between them. Indeed, if an algorithm can select a set of materialized views without considering subquery relations but leads to the same query efficiency as our AODC-based view selection algorithm (Alg. 4), Alg. 4 is unnecessary.

We consider the algorithm that materializes the workload queries in the descending order of frequency until exceeding the memory budget as the representative of materialized view selection algorithms that disregard subquery relations. In the following, we call this algorithm the no-subquery algorithm. We aim to compare its output materialized view set's efficiency with the best

[2]Our implementation resides at https://github.com/pkumod/rpq-view.

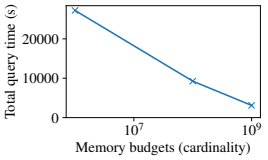


(a) Original distribution.

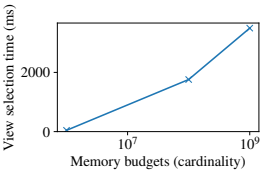


(b) Deduplicated distribution.

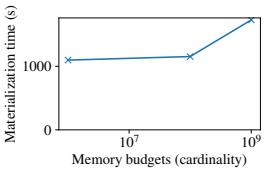
Fig. 7. The speedup distribution of AODC plans compared with minimal DFA plans.



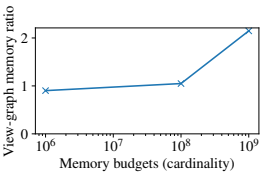
(a) Query time.



(b) View selection time.



(c) Materialization time.



(d) View-graph memory ratio.

Fig. 8. The impact of the memory budget on performance.

output of Alg. 4, i.e., with the memory budget equal to 10^9. Therefore, we set a hard memory budget for the no-subquery algorithm in terms of the number of gigabytes used in the actual run of Alg. 4, which is 131GB, instead of the estimated cardinality. We then compare the workload's total query time using the no-subquery algorithm's output set of materialized views and ours.

The total query time using the materialized views selected by Alg. 4 is 3102.02 s, while that of using the no-subquery algorithm is 6661.38 s. The speedup is 2.15×. In addition, Alg. 4 also selects more materialized views (206 views) than the no-subquery algorithm (70 views), which is due to Alg. 4 making use of the cardinality estimation scheme and skipping views that can cause exceeding the memory budget. Such a strategy skips the views with a low speedup-memory ratio, further enhancing the query efficiency.

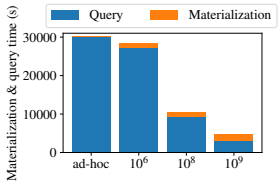
8.4 Impact of the Memory Budget

In this section, we test the impact of the memory budget (b in Alg. 4) on the efficiency of MVS and view-based querying. The Monte Carlo sampling size (Sec. 7.2.5) is fixed as 10, since it does not significantly impact the performance within $[10, 100]$. Fig. 8 shows the query time, view selection time, materialization time, and view-graph memory ratio varying b across 10^6, 10^8, and 10^9.

8.4.1 Materialization Time & View-Graph Memory Ratio. These have the same growth trend as the memory budget increases. Though the memory budget grows exponentially, the actual memory usage grows almost linearly due to underestimating the cardinalities of some views. The underestimation seems counter-intuitive, since we emphasize the possibility of overestimation in Sec. 7.2.2 and 7.2.3 due to ignoring the overlap between result sets that are to be merged. We conjecture that the underestimation is due to the uniformity assumption, which we plan to relax in the future work.

8.4.2 Query Time. The workload's total query time achieves linear speedup with the actual memory usage. The speedup is 9.73× when the budget = 10^9, close to an order of magnitude, showing the effectiveness of the selected materialized views in querying.

8.4.3 View Selection Time. The view selection time also grows with the memory budget, since it mainly consists of the incremental planning (Alg. 5) time, and a small budget skips the views with



(a)

Budgets	Q_a/Q_v	$Q_a/(Q_v+M)$
10^6	1.11×	1.07×
10^8	3.26×	2.90×
10^9	9.73×	6.25×

(b)

Table 4. Impact of the workload size.

Workload size	Q_a/Q_v	$Q_a/(Q_v+M)$
100	3.85	2.86
10	7.71	2.56

Fig. 9. View-based and ad-hoc querying comparison.

large estimated cardinalities, thus reducing the invocations of Alg. 5. However, note that the view selection time is almost negligible in the overall workflow: Fig. 8b shows it in microseconds.

8.4.4 Comparison with Ad-Hoc Processing. Though we conduct the view selection and materialization offline, their running time still needs to be within a reasonable range. To inspect the proportion between the materialization time and the query time, we plot them in Fig. 9 with varying memory budgets and contrast them with ad-hoc processing, which does not use any materialized views.

Fig. 9a shows that materialization is always much faster than querying, and that view-based processing is faster than ad-hoc processing regardless of the memory budget. Fig. 9b provides the detailed statistics of the comparison between the ad-hoc and view-based methods. Q_a and Q_v are the ad-hoc and view-based query time, respectively, and M is the materialization time. The second and third columns thus show the speedup without and with the materialization time. A larger budgets causes the speedup to decrease more with the materialization time. However, even with the materialization time, the speedup still grows significantly with the budget and reaches an impressive 6.25× with $b = 10^9$.

We show the speedup of RPQs of different patterns in Tab. 5, classifying patterns by the operator sequence as in [5]. Due to limited space, we only show the top-8 patterns, accounting for >60% of distinct RPQs in the workload. RPQs with Kleene closures have greater speedup than the others, since Kleene closures are usually more expensive to execute, thus benefiting more from materialized views. The speedup of the concatenation and Kleene closure operators compounds, as observed with ()*/()*. RPQs with alternation only are hardly sped up because their subqueries are rarely materialized, given how close their execution costs are to their result sizes.

8.5 Impact of the Workload Size

To show our method's effectiveness on workloads of different sizes, we sample 5 workloads with 100 and 10 RPQs respectively from the full workload, choosing each RPQ independently at random. Tab. 4 shows our method's speedup compared with ad-hoc processing when $b = 10^8$, which remains significant despite decreasing with the workload size due to fewer overlapping queries.

8.6 Comparison with State-of-the-Art

In this section, we compare our method with the state-of-the-art MQO methods for RPQs, RTC [15] and Swarmguide [1], since there are no MVS methods to compare with. For fairness, we consider their optimization time as the offline time, only comparing the workload's total query time and the memory usage. We also compare with the state-of-the-art ad-hoc RPQ methods, Ring [5] and Unit [18], to show the necessity and effectiveness of MVS for RPQ.

8.6.1 Implementation of Competitors. Ring's code is provided by its authors. We reimplement the other competitors as follows:

Table 5. Speedup of RPQ patterns w.r.t. ad-hoc processing.

Pattern	%	Speedup
/	12.44	2.91
/()*	11.54	6.33
|	7.69	1.03
()*	7.69	10.63
-	5.88	1.34
()+	3.85	6.65
()*/()*	2.26	236.15
||	2.04	0.99

Table 6. Comparison with state-of-the-art.

Method	Q. time (s)	Speedup	Mem. Ratio	Speedup-Mem. Ratio
Ours	3102.02	9.73	2.15	4.53
RTC [15]	27090.30	1.11	1.72	0.65
Swarmguide [1]	31857.30	0.95	1.63	0.58
Ring [5]	904823.70	0.03	-	-
Unit [18]	80671.07	0.37	-	-

- RTC: RTC materializes the shared Kleene closures of workload RPQs. Hence, we implement it by only considering candidate views that are Kleene closures in Alg. 4, setting $b = 10^9$. Since our work addresses MVS and view-based query planning for RPQs, we compare our method with RTC only in this regard, while its efficient physical implementation of the Kleene closure operator will be an add-on to our framework in the future (Sec. 4).
- Swarmguide: We define the similarity between two DFAs used in affinity propagation as $\frac{|\Sigma_1 \cap \Sigma_2|}{|\Sigma_1 \cup \Sigma_2|}$, where Σ_1 and Σ_2 are the DFAs' label sets, respectively, according to [1]. Since [1] does not give the damping factor for affinity propagation, we try 0.2, 0.4, 0.6 and 0.8, where 0.8 yields the highest speedup-memory ratio (Tab. 6). After obtaining the selected views, we plug them into the AODC and invoke Alg. 5 for incremental planning, instead of using Waveguide as in [1]. Sec. 8.6.2 will discuss its implications.
- Unit: We enumerate all the ways of splitting an RPQ fixing the maximum number of threads as four, since most workload RPQs have fewer than four concatenation operators. Since [18] cannot estimate the cost of unbounded Kleene closures, we force an upper bound of six steps on all the Kleene closures during Unit's cost estimation. To ensure the accuracy of the query results, we keep the Kleene closures unbounded during Unit's execution. The number of threads spanned for execution is equal to the number of subqueries in the chosen batch.

8.6.2 Result Analysis. Tab. 6 gives the workload's total query time, the speedup compared with ad-hoc processing, the view-graph memory ratio, and the speedup-memory ratio of our method and the competitors. The speedup-memory ratio is the result of dividing the speedup by the memory ratio. The higher a method's speedup-memory ratio, the more efficient its selected views utilize memory. Ad-hoc methods do not have the memory-related metrics, since they do not use materialized views.

Overall, our method achieves the highest speedup and speedup-memory ratio, though it uses more memory than its competitors.

RTC has lower speedup, memory, and speedup-memory ratio than our method. The reason is that though Kleene closures are difficult to process, which means materializing them will save much query time, their results are usually large. Therefore, not many Kleene closures can be materialized within a memory budget. Meanwhile, other subqueries that are not Kleene closures but are more cost-efficient to materialize are ignored.

Swarmguide shows the same trend as RTC. Its total query time is roughly the same as the ad-hoc processing time, meaning the materialized view that it selects are hardly useful on this workload. The reason is twofold. Firstly, though the workload queries in single-element clusters are directly materialized, they are not guaranteed to be efficient. Secondly, the clusters with multiple elements commonly have a small or even empty maximum common automaton, because the label-based

affinity propagation clustering does not consider the topological structure of DFAs. For example, it may cluster the queries a/b, b/c, and c/a together, since the propagation procedure detects the relation among the labels a, b, and c, but their maximum common sub-automaton is empty.

Ring and Unit are slower than our ad-hoc method using AODC plans. This does not contradict the results reported in their papers, since [5] does not include queries whose execution time exceeds 60 seconds when collating the average query time and [18] forces all the Kleene closures to be bounded during execution. Ring is slow because it is not specifically optimized for RPQs without constant source or target nodes; Unit is slow because it does not use sideways information passing, which is difficult to implement in parallel.

Lastly, we discuss the implications of plugging the materialized views selected by Swarmguide into the AODC instead of Waveplans. It is possible for Swarmguide to select a materialized view that the AODC cannot use due to not supporting no LC and partial LC plans. For example, Swarmguide may materialize $a/b/c$ when given the cluster $\{a/b/c, a/(b/c)^+\}$, but the AODC cannot use it during $a/(b/c)^+$'s execution, since using it requires partial LC. However, out of the 21 materialized views selected by Swarmguide in this experiment, only 1 belongs to this case. Therefore, using Waveguide or supporting no LC and partial LC on the AODC is unlikely to significantly improve Swarmguide's performance on this workload.

9 CONCLUSION

In this paper, we propose a framework for materialized view selection (MVS) and view-based query planning for RPQs. We derive the AND-OR DAG with closure (AODC) to represent an RPQ workload's query plan and design cost and cardinality estimation schemes for query planning. We show the equivalence between the plan spaces of the AODC and Waveplans [27] except that the AODC does not support no LC and partial LC strategies for Kleene closures, which rarely improve the query efficiency.

Since MVS for RPQs is hard, we design a heuristic MVS algorithm and use the AODC to detect and remove redundant views. We also propose an incremental view-based planning algorithm that efficiently updates the AODC as views are newly selected, so that the view-based plan is immediately available after MVS. Experiments show that view-based RPQ processing outperforms state-of-the-art ad-hoc methods [5, 18] on a workload and has a higher cost-performance ratio than MQO methods [1, 15].

In the future, we plan to incorporate efficient physical implementations of the Kleene closure, such as RTC's [15]. We also plan to improve the cardinality estimation of RPQs by tightening the relaxation in Eqn. 26 and removing the uniformity assumption. In addition, we will study how to efficiently apply MVS for RPQ techniques in the context of graph DBMS, especially optimizing the execution order of multiple RPQs and other query constructs.

ACKNOWLEDGMENTS

This work was supported by The National Key Research and Development Program of China under grant 2023YFB4502303, the Research Grants Council of Hong Kong, China, No.14205520, and NSFC grants under U20A20174, 61932001. Lei Zou is the corresponding author of this paper.

REFERENCES

[1] Zahid Abul-Basher. 2017. Multiple-Query Optimization of Regular Path Queries. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, San Diego, CA, USA, 1426–1430. https://doi.org/10.1109/ICDE.2017.205

[2] Zahid Abul-Basher, Parke Godfrey, Nikolay Yakovets, and Mark Chignell. 2016. SWARMGUIDE: Towards Multiple-Query Optimization in Graph Databases. (2016).

[3] Sergey Afonin. 2008. The View Selection Problem for Regular Path Queries. In *LATIN 2008: Theoretical Informatics*, Eduardo Sany Laber, Claudson Bornstein, Loana Tito Nogueira, and Luerbio Faria (Eds.). Vol. 4957. Springer Berlin Heidelberg, Berlin, Heidelberg, 121–132. https://doi.org/10.1007/978-3-540-78773-0_11

[4] 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.). Springer Nature Switzerland, Cham, 38–54.

[5] Diego Arroyuelo, Aidan Hogan, Gonzalo Navarro, and Javiel Rojas-Ledesma. 2022. Time- and Space-Efficient Regular Path Queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, Kuala Lumpur, Malaysia, 3091–3105. https://doi.org/10.1109/ICDE53745.2022.00277

[6] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2024. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *Comput. Surveys* 56, 2 (Feb. 2024), 1–40. https://doi.org/10.1145/3604932

[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] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An Analytical Study of Large SPARQL Query Logs. *The VLDB Journal* 29, 2-3 (May 2020), 655–679. https://doi.org/10.1007/s00778-019-00558-9

[9] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 2246–2258. https://doi.org/10.1145/3514221.3526057

[10] Saumen Dey, Víctor Cuevas-Vicenttín, Sven Köhler, Eric Gribkoff, Michael Wang, and Bertram Ludäscher. 2013. On implementing provenance-aware regular path queries with relational query engines *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 214–223. https://doi.org/10.1145/2457317.2457353

[11] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying. In *EDBT*. 520–523.

[12] Ina Koch. 2001. Enumerating all connected maximal common subgraphs in two graphs. *Theoretical Computer Science* 250, 1 (2001), 1–30. https://doi.org/10.1016/S0304-3975(00)00286-3

[13] André Koschmieder and Ulf Leser. 2012. Regular Path Queries on Large Graphs. In *Scientific and Statistical Database Management*, Anastasia Ailamaki and Shawn Bowers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 177–194.

[14] 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.

[15] Inju Na, Yang-Sae Moon, Ilyeop Yi, Kyu-Young Whang, and Soon J. Hyun. 2022. Regular Path Query Evaluation Sharing a Reduced Transitive Closure Based on Graph Reduction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, Kuala Lumpur, Malaysia, 1675–1686. https://doi.org/10.1109/ICDE53745.2022.00171

[16] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. 1978. An analysis of approximations for maximizing submodular set functions–I. *Math. Program.* 14, 1 (dec 1978), 265–294. https://doi.org/10.1007/BF01588971

[17] M. E. J. Newman. 2000. Models of the Small World. *Journal of Statistical Physics* 101, 3/4 (2000), 819–841. https://doi.org/10.1023/A:1026485807148

[18] Van-Quyet Nguyen, Quyet-Thang Huynh, and Kyungbaek Kim. 2022. Estimating Searching Cost of Regular Path Queries on Large Graphs by Exploiting Unit-Subqueries. *Journal of Heuristics* 28, 2 (April 2022), 149–169. https://doi.org/10.1007/s10732-018-9402-0

[19] Maurizio Nolé and Carlo Sartiani. 2016. Regular Path Queries on Massive Graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management* (Budapest, Hungary) *(SSDBM '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 12 pages. https://doi.org/10.1145/2949689.2949711

[20] Nicholas Roussopoulos. 1982. View indexing in relational databases. 7, 2 (jun 1982), 258–290. https://doi.org/10.1145/319702.319729

[21] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (Dallas, Texas, USA) *(SIGMOD '00)*. Association for Computing Machinery, New York, NY, USA, 249–260. https://doi.org/10.1145/342009.335419

[22] Jacob Scott, Trey Ideker, Richard M. Karp, and Roded Sharan. 2006. Efficient Algorithms for Detecting Signaling Pathways in Protein Interaction Networks. *Journal of Computational Biology* 13, 2 (2006), 133–144. https://doi.org/10.1089/cmb.2006.13.133

[23] Lucien D.J. Valstar, George H.L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 345–358. https://doi.org/10.1145/3035918.3035955

[24] Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: A Free Collaborative Knowledgebase. *Commun. ACM* 57, 10 (Sept. 2014), 78–85. https://doi.org/10.1145/2629489

[25] Sarisht Wadhwa, Anagh Prasad, Sayan Ranu, Amitabha Bagchi, and Srikanta Bedathur. 2019. Efficiently Answering Regular Simple Path Queries on Large Labeled Networks. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1463–1480. https://doi.org/10.1145/3299869.3319882

[26] Xin Wang, Junhu Wang, and Xiaowang Zhang. 2016. Efficient Distributed Regular Path Queries on RDF Graphs Using Partial Evaluation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (Indianapolis, Indiana, USA) *(CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 1933–1936. https://doi.org/10.1145/2983323.2983877

[27] 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

Received October 2023; revised January 2024; accepted February 2024