Multi-query optimization for entity-to-entity path discovery in data journalism

INF517 Intermediate Report

Jean Law
advised by Ioana Manolescu

CEDAR team
Inria Saclay and LIX
Ecole Polytechnique
3 January 2023

Contents

1 Introduction **2**
1.1 Problem statement . . . 2

2 State-of-the-art in MQO **3**
2.1 Related work . . . 3
2.2 General observations about MQO . . . 4

3 Useful properties of ConnectionLens path queries (pqueries) **4**
3.1 First definitions and properties . . . 4
3.2 Connected subqueries . . . 6
3.3 Most specific subqueries . . . 7

4 Proposed algorithms for MQO on a set of pqueries **8**
4.1 Algorithm 1: Identifying the most specific subqueries of two pqueries . . . 8
4.2 Algorithm 2: Greedy MQO algorithm . . . 11
4.3 Quantitative measure of selectivity or cost . . . 11
4.4 Experiments in progress . . . 12

5 Annexes **13**
5.1 Annex 1: Preliminary notions . . . 13
5.1.1 Query languages and query dialects . . . 13
5.1.2 Query semantics . . . 14
5.1.3 Relational algebra vs. conjunctive queries . . . 15
5.1.4 Query containment and equivalence . . . 16
5.1.5 Selectivity . . . 16
5.2 Annex 2: Details of Scalable Multi-Query Optimization for SPARQL . . . 17
5.2.1 Bootstrapping . . . 17
5.2.2 Refining query clusters with hierarchical clustering . . . 17
5.2.3 Finding the Maximum Common Edge Subgraphs between two clusters . . . 17

1 Introduction

ConnectionLens [CDG+18, ABB+21, AMM22] is a system designed for data journalism that integrates heterogeneous, independently authored data sources in a single graph. The nodes in ConnectionLens represent data items that initially originated from various data structures such as relational tuples, RDF, JSON or XML nodes, whereas iits edges correspond either to structural connections that were already existed within the data source (e.g. a tuple is in a database, an attribute is in a tuple, a JSON node has a parent) or to similarity (sameAs) links between sources. ConnectionLens is particularly interesting for investigative journalists as it allows them to explore a large number of datasets simultaneously, as well as automatize the process of finding similarities and connections between entities mentioned in these datasets.

ConnectionLens comes with a dataset abstraction tool, Abstra [BMU22] that automatically generates a high-level description of the dataset. One important feature of Abstra is that it classifies each node in the ConnectionLens graph into categories, such as organization, location, person, that help the user to understand the nature of the objects in the dataset. These categories are known as **collection entities**. Afterwards, Abstra summarizes all collection paths that are present in the graph (i.e. all paths between collection entities in the ConnectionLens graph) in the form of a collection graph.

This project is a continuation of the work of a previous intern, Antoine Gauquier (IMT Lille), who developed a new way to explore a ConnectionLens graph during the summer of 2022. His work enumerates all collection paths that connect an entity of type A with an entity of type B, where A and B are the same or distinct collection entities. For instance, if A is Person and B is Organization, one path in the collection graph may be "Person$\xrightarrow{\text{worksFor}}$Organization", while another path may be "Person$\xrightarrow{\text{livesIn}}$Place$\xleftarrow{\text{hasHeadquartersIn}}$Organization". It is worth noting that some of the collection paths returned by this query may not actually exist in the graph. For instance, the collection path "Person$\xrightarrow{\text{livesIn}}$Place$\xleftarrow{\text{hasHeadquartersIn}}$Organization" might have been generated exclusively by two distinct paths on the ConnectionLens graph, such as "Johnny (Person) $\xrightarrow{\text{livesIn}}$Paris (Place)" and "New York (Place)$\xleftarrow{\text{hasHeadquartersIn}}$ Apple(Organization)". In this case, there is no real path on the ConnectionLens graph that corresponds to this collection path.

These collection paths must additionally fit a set of criteria defined by the user, including the maximum path length, presence of inversions (explained in Section 1.1) and the exclusion of specified intermediate named entity types.

After the enumeration of these collection paths, the actual data paths on the ConnectionLens graph that they correspond to need to be efficiently computed. This may be challenging for many reasons, such as:

- A long path corresponds to a complex query.
- If the graph is large, even a short path may have many answers.
- Several paths may share some segments. Computing them multiple times wastes computing power.
- Some paths have no answers. Detecting those as early as possible is important.

The goal of this project is hence to develop an efficient way algorithm that computes a set of Entity-to-Entity paths present in a ConnectionLens graph.

1.1 Problem statement

We are given a set of acyclic and connected ConnectionLens **path queries** (defined in Section 3) $Q = \{q_1, ..., q_n\}$ with $n \geq 2$. Each ConnectionLens path query corresponds to a collection path, and all of the collection paths start (resp. end) with the same collection entity, i.e. given two collection entities A and B, each path begins with a node of collection entity A and ends with a node of collection entity B.

The objective of this project is to devise an algorithm that computes the results to the queries in Q more efficiently than directly executing each query in Q. We have narrowed down the focus of this project to identifying paths that share the same subpaths and ensuring that these shared subpaths are only computed

once (as opposted to every time they are needed). This is a Multi-Query Optimization (MQO) problem, and the algorithm should produce an output set of queries $O = \{q_1, ..., q_m\}$ in which $m \geq n$ and, for each q_i in Q, there exists a combination of path queries in O that, when combined will give us q_i.

After computing O, the algorithm should return the results to the n queries in Q. To do this, it needs to compute a directed acyclic graph (DAG) that details how the queries in O should be combined to produce the results in Q. Afterwards, the algorithm should combine the queries in O according to the DAG to produce the desired results.

The algorithm must also ensure that paths that contain *inversions* are computed correctly. Inversions are illustrated in the following example:

- From a Person node identified in a node n_1
- To a company n_2 that n_1 worked for: $n_1 \xrightarrow{\text{worksFor}} n_2$
- To another person n_3 that worked in the same company: $n_2 \xleftarrow{\text{worksFor}} n_3, n_1 \neq n_3$
- To the place where n_3 was born: $n_3 \xrightarrow{\text{bornIn}} n_4$
- Such that a Location node has been extracted from n_4.

The above is a path with one inversion, that goes from a Person entity to a Location entity. Note the inequality predicate $n_1 \neq n_3$ that comes with the inversion.

2 State-of-the-art in MQO

2.1 Related work

In order to understand the state-of-the-art in MQO, we refer to the following two articles. The first proposes a MQO algorithm for relational relational queries, whereas the second optimizes jointly multiple SPARQL queries on RDF graphs.

In Improvements on a Heuristic Algorithm for Multi-Query Optimization [SSN94], K. Shim et al. use a heuristic search algorithm to optimize n relational queries, q_1, ..., q_n. Each query q_i, $1 \leq i \leq n$, needs to be computed using a relational query access plan, and each access plan is made up of operations. Each operation has a cost, and the cost of each access plan is the sum of costs of all of its operations that have not already been used in the access plans of q_1,...,q_{i-1}. In order to produce a globally optimal access plan that minimizes the global cost, the team uses a A*state-space search algorithm with a heuristic function h_n that, at each node in the branch-and-bound procedure, estimates the cost of the remaining queries. K. Shim et al.'s proposed heuristic function improves the heuristic function that was initially suggested by Sellis in 1994 [Sel88] by recalculating the value of h_n at each node explored, and taking into account, for each plan of q_i, whether each of its operation has already been used in q_1,...,q_{i-1}, and if not, how many times it appears in the plans of q_{i+1},...,q_n. With this new heuristic function, a smaller fraction of nodes are expanded by the algorithm.

On the other hand, in Scalable Multi-Query Optimization for SPARQL [LKDL12], Le et al. have implemented a heuristic search algorithm for this problem. In this paper, the goal is to turn a set of SPARQL queries that are either "of type 1" (strictly conjunctive) or "of type 2" (conjunctive and some optional parts) into another set of queries, such that the evaluation of the second set of queries exploits commonalities between the queries and reduces their global evaluation costs.

To identify sharing opportunities, given that they represent queries as graphs, they should (in principle) find isomorphic subgraphs among every pair of query graphs. However, this is expensive (finding isomorphisms between 2 graphs is NP hard in the size of the graphs; further, in this setting, they should find isomorphisms between pairs of *subgraphs*, of which there may be many). Therefore, they use a set of heuristic algorithms in order to optimize the queries in an efficient way. First, they apply k-means on the input batch of queries, using the Jaccard similarity as the similarity metric, to partition it into groups such that each group of queries can be optimized together (Section 5.2.1). Afterwards, for each partitioned group of queries, they further merged

structurally similar queries by using hierarchical clustering (Section 5.2.2), with the similarity between two clusters of queries being calculated by constructing the maximal common edge subgraphs (Section 5.2.3).

2.2 General observations about MQO

All MQO problems are defined by an input set of queries and an output set of queries, and the goal is to find what to evaluate so that some subqueries from the input are evaluated only once (as opposed to every time they are needed), thus reducing the overall evaluation cost.

Each MQO algorithm makes some choices:

- What is the "sharing unit"? The relational MQO paper shares **tasks** which are operations in the relational query plans. In contrast, the SPARQL MQO paper shares **subqueries** which are subgraphs of the original queries.
- What are we allowed to evaluate? The relational MQO paper only evaluates tasks that exist in at least one of the input queries. The SPARQL paper can evaluate queries that did not exist in the input.
- How to enumerate sharing units (sharing opportunities)? The relational MQO paper uses an A* version. The SPARQL paper uses a set of heuristics (including clustering and Maximal Common Edge subgraphs) plus their own techniques.
- How to estimate what is the most cost-efficient sharing? We may have several sharing opportunities, each of which reduces the evaluation cost in a certain way. The relational paper assumes given a cost for each task, and then introduces a specialization of the A* cost function for their problem. The MQO paper relies on some selectivity and cost estimation approach from prior work ([33]).

Compared to the literature, our problem (MQO on paths) is

- simpler than the SPARQL work [LKDL12] because all our queries are guaranteed to have a path structure
- potentially more complicated by the presence of inversions that bring inequality predicates.

3 Useful properties of ConnectionLens path queries (pqueries)

We first establish some definitions and properties on ConnectionLens path queries which will be useful in proving the correctness of our proposed algorithm. We notably establish that the optimal sharing units that we want to use to optimize the evaluation cost are syntactically valid pqueries.

3.1 First definitions and properties

We model a ConnectionLens graph as two relations, storing respectively nodes and edges. Specifically, we use $n(\underline{id}, \tau, l, eq)$ for nodes, where τ is the type of a node, l is its label (possibly empty , denoted ϵ), and eq identifies an equivalence class to which the node belongs (we can think of eq as an integer). The edge relation is $e(s, t, l)$, where s and t are foreign keys in the nodes (n) relation, thus they are node IDs, and l is the label of the edge.

Definition 1 (ConnectionLens path query) *A ConnectionLens path query (or just path query, or pquery, in short) is a conjunctive query over the relations n, e of the form:*

$$q(\bar{x}) :\text{-} an_1, ae_1, an_2, ae_2, \ldots, an_{k-1}, ae_{k-1}, an_k$$

where k is a positive integer; each an_i, for $1 \leq i \leq k$, is an atom referring to the $\underline{n}$ relation, with a constant in the equivalence class position; each ae_j, for $1 \leq j < k$, is an atom referring to the $\underline{e}$ relation; and $\bar{x}$ is a subset of the variables in the body of the query.

Example 1 *The following is a sample query for $k = 3$:*

$$pq_1(x) \text{ :- } \quad n(z, XML_NODE, 'book', \underline{1}), e(z, u, \epsilon), n(u, XML_NODE, 'title', \underline{2}), \\ e(u, v, \epsilon), n(v, XML_VALUE, x, \underline{3})$$

For readability, we will always associate underlined constants, e.g., $\underline{1}$ etc., to IDs of equivalence classes.

Definition 2 (Subquery) *Given a pquery pq whose head variables are $\bar{x}$, a* subquery *of pq is a CQ sq_1 such that there exists another pquery sq_2, and a selection condition σ_{sel} such that:*

$$pq \equiv \pi_{\bar{x}}(\sigma_{sel}(sq_1 \bowtie sq_2))$$

where $sq_1 \bowtie sq_2$ denotes the natural join of sq_1 and sq_2 on all their shared variables (recall Section 5.1.3).

As particular cases, in the above: $\pi_{\bar{x}}$ may not be needed (if its input returns already the right attributes); σ_{sel} may not be needed (if there is no extra condition); sq_2 may be empty, in which case $sq_1 \bowtie sq_2$ becomes just sq_1.

Example 2 *Given the pquery pq_1 from Example 1, a possible proper subquery sq_1, corresponding sq_2, and equivalent rewriting of pq_1 based on sq_1 and sq_2 are:*

$$\begin{aligned} sq_1(z, u, v, x) \text{ :- } & \quad n(z, XML_NODE, l, \underline{1}), n(u, XML_NODE, 'title', \underline{2}), n(v, XML_VALUE, x, \underline{3}) \\ sq_2(z, u, v) \text{ :- } & \quad e(z, u, \epsilon), e(u, v, \epsilon) \\ pq_1(x) \text{ :- } & \quad \pi_x(\sigma_{l='book'}(sq_1 \bowtie sq_2)) \end{aligned}$$

Note that each sq_1 is a cartesian product of three instances of the n relation. This shows that even if a pquery is connected (each atom shares at least one variable with other atoms), its subqueries are not necessarily connected. Below, we generalize this observation to derive the set of all subqueries of a given pquery.

Definition 3 (Canonical form of a pquery) *Let $q(\bar{x})$:- $an_1, ae_1, an_2, ae_2, \ldots, an_{k-1}, ae_{k-1}, an_k$ be a pquery. We call the* canonical form of q, *denoted $q^c(\bar{x})$, the expression $\sigma_c(q_0(\bar{x}))$, where*

- *$q_0(\bar{x})$:- $n(v_1^1, v_1^2, v_1^3, \underline{c_1}), e(u_1^1, u_1^2, u_1^3), n(v_2^1, v_2^2, v_2^3, \underline{c_2}), \ldots, e(u_{k-1}^1, u_{k-1}^2, u_{k-1}^3), n(v_k^1, v_k^2, v_k^3, \underline{c_k})$ is a CQ*
  - *whose body contains k atoms over the n table, and $k-1$ atoms over the e table;*
  - *in each of these atoms, the first three attributes are variables, while the last attribute is the constant equivalence class, appearing in the corresponding position in q;*
  - *no variable appears more than once, that is, all the atoms participate in a cartesian product.*
- *σ_c is a selection with all the necessary conditions so that $\sigma_c(q_0(\bar{x}))$ is exactly q.*

Example 3 (Canonical form of a pquery) *Considering the pquery pq_1 from Example 1, we obtain:*

- *$pq_0(x)$:- $n(v_1^1, v_1^2, l, \underline{1}), e(u_1^2, u_1^2, u_1^3), n(v_2^1, v_2^2, v_2^3, \underline{2}), e(u_2^1, u_2^2, u_2^3), n(v_3^1, v_3^2, x, \underline{3})$ (for readability, we left unchanged the names of each variable that appears only once in pq_1)*
- *σ_c is: $(v_1^1 = u_1^2) \wedge (v_1^2 = XML_NODE) \wedge (l =' book') \wedge (u_2^1 = v_1^1) \wedge (u_1^3 = \epsilon) \wedge (v_2^2 = XML_NODE) \wedge (v_2^3 =' title') \wedge (v_2^1 = u_2^1) \wedge (u_2^3 = \epsilon) \wedge (u_2^2 = v_3^1) \wedge (v_3^2 = XML_VALUE)$ (the condition is a conjunction of 11 predicates).*

Notation We say that a pquery q is **defined by** $\langle \bar{x}, \sigma_c, q_0 \rangle$, where $\bar{x}$ denotes the head variables of q, while σ_c and q_0 are components of the canonical form of q.

Property 1 (Subqueries of a pquery) *Let q be a pquery such that q_0 and σ_c compose the canonical form of q.*

1. *Let σ_b be a subset of the predicates in σ_c. Then, the CQ sq_1 defined as $\sigma_b(q_0)$ is a subquery (Definition 2) of q.*
2. *Let b be a subset of the atoms in the body of q_0, σ_b be a subset of the predicates in σ_c that apply only on variables present in b, and $\bar{y}$ be the set of q variables that includes $\bar{x}$ and any other variable v such that σ_c contains a predicate of the form $v = x$ where $x \in \bar{x}$ and $v \notin \bar{x}$.*
   Let q_b be the query whose head is $\bar{y}$ and body consists exactly of the atoms in b.
   Let sq_1 be the CQ whose canonical form is as $\pi_{\bar{x}}(\sigma_b(q_b))$. Then, sq_1 is a subquery of q.

Proof

1. Let σ_d be the conjunction of all predicates in σ_c that are not in σ_b. It is easy to see that $q \equiv \sigma_d(sq_1)$.
2. Let d be the set of atoms in the body of q that are not in b, and σ_d be all the σ_c predicates that are not in σ_b. Let $\bar{z}$ be the set of all variables appearing in d and/or σ_d. Let q_d^1 be the CQ having the head $\bar{z}$ and the body consisting of exactly the atoms of d; let $q_d^2 = \sigma_d(q_d^1)$. It is easy to see that $q \equiv \pi_{\bar{x}}(sq_1 \bowtie sq_2)$.

Property 1 provides **a way to enumerate subqueries of a query** q_0: select σ_b, then b and $\bar{x}$ as prescribed, and obtain a subquery q_1.

From Property 1, it follows that a the number of subqueries of q_0 is at least $O(2^{|\sigma_c|}) + O(2^{|q|})$, where $|\sigma_c|$ denotes the number of predicates in σ_c and $|q|$ denotes the number of atoms in the body of q. Some of these subqueries may be isomorphic to each other (and, thus, logically, should count as one). For instance, applying Property 1 point 2. to the sample query pq_1 and its canonical form shown in Example 3, the subquery sq_1' obtained with $\sigma_b = \emptyset$ and the first e atom, respectively, sq_1'' obtained with $\sigma_b = \emptyset$ and the second e atom are isomorphic. However, even taking this into account, the number of subqueries of a pquery remains quite high.

At the same time, *some subqueries may not be connected*, i.e., include cartesian products. This is the case, for instance, of sq_1 and sq_2 in Example 2. We are not interested in such subqueries, since their evaluation can hardly be made efficient, and (especially) their result size is likely to be huge. Therefore, evaluating such a subquery is unlikely to help our performance. Instead, we focus on *connected subqueries*, as follows:

3.2 Connected subqueries

In this subsection, we prove that connected subqueries, which are the sharing units of this MQO problem that will be constructed in the algorithm, are valid pqueries (in terms of syntax).

Definition 4 (Connected subquery) *Given a pquery q whose canonical form is defined by q_0 and σ_c, and having the head variables $\bar{x}$. A connected subquery s of q is a CQ having the following canonical form:*

- *The body of s, denoted b, is a* subsequence *of the atoms in the body of q_0, starting and ending with an n atom;*
- *Let σ_s be a subset of the predicates in σ_c that only refer to the variables in b, which, if $|b| > 1$, ensures that each atom in b is connected (shares a variable with) at least another atom in b;*
- *Let $\bar{y}$ be the variables appearing in b that either (i) appear in $\bar{x}$, or (ii) appear in some predicate in $\sigma_c \setminus \sigma_s$.*
- *Then, s is the CQ defined by the canonical form $\pi_{\bar{y}}(\sigma_s(b))$.*

Example 4 *We continue to to rely on the sample query pq_1 and its canonical form shown in Example 3.*

- *Let b consist of the last three atoms $n(v_2^1, v_2^2, v_2^3, \underline{2}), e(u_2^1, u_2^2, u_2^3), n(v_3^1, v_3^2, x, \underline{3})$ from the body of pq_0.*
- *The predicates in σ_c that only refer to these variables are:*

$$(v_2^2 = XML_NODE) \wedge (v_2^3 =' title') \wedge (v_2^1 = u_2^1) \wedge (u_2^3 = \epsilon) \wedge (u_2^2 = v_3^1) \wedge (v_3^2 = XML_VALUE)$$

- *We can, for instance, take this whole set as σ_s. It contains two predicates $(v_2^1 = u_2^1)$ and $(u_2^2 = v_3^1)$ which ensure that the atoms in b are connected.*
- *Then, $\bar{y}$ becomes: (x, u_2^1). We added u_2^1 because $\sigma_c \setminus \sigma_s$ contains the predicate $(u_2^1 = v_1^2)$.*
- *The resulting subquery s has the canonical form: $\pi_{\bar{y}}(\sigma_s(b))$, which corresponds to:*

$$s(x, u_2^1) :- n(v_2^1, XML_NODE, 'title'), e(u_2^1, u_2^2, \epsilon), n(u_2^2, XML_VALUE, x)$$

Compared with the original pquery (Example 1), it corresponds to: taking the last three atoms; renaming some variables; and adding to the head, the variable that s shares with the atoms in pq_1's body, that are not in the body of s.

Property 2 *Any connected subquery (Definition 4) is a pquery (Definition 1).*

Proof We show that a connected subquery s (of the pquery q defined by the canonical form q_0, σ_c) is a conjunctive query over the relations n, e of the form:

$$\mathrm{q}(\bar{x}) \text{:-} an_1, ae_1, an_2, ae_2, \ldots, an_{k-1}, ae_{k-1}, an_k$$

where k is a non-zero integer; each an_i, for $1 \leq i \leq k$, is an atom referring to the n relation; each ae_j, for $1 \leq j < k$, is an atom referring to the e relation; and $\bar{x}$ is a subset of the variables in the body of the query.

s, by definition, is a conjunctive query. It is a connected subquery, so there exists a subsequence of q_0 that starts and ends with an n atom that defines its body that we denote by b. Additionally, σ_s ensures that each atom in b is connected with another atom in b. This proves that s has the aforementioned form.

Property 3 *Let pq be a pquery (Definition 1) defined by $\langle \bar{x}, q_0, \sigma_c \rangle$. Any connected subquery s (Definition 4) of pq is also a subquery (in the sense of Definition 2) of pq.*

Proof We show that s is a CQ sq_1 such that there exists another pquery sq_2, and a selection condition σ_{sel} such that:

$$pq \equiv \pi_{\bar{x}}(\sigma_{sel}(sq_1 \bowtie sq_2))$$

where $sq_1 \bowtie sq_2$ denotes the natural join of sq_1 and sq_2 on all their shared variables.

Property 2 ensures that q_1 is a pquery. Assume it is determined by $\langle \bar{y_1}, b_1, \sigma_{c_1} \rangle$. b_1 is a subsequence of q_0 hence there exists b_2 such that $b_1 \cup b_2 = q_0$. We define the canonical form of sq_2 as b_2 and $\sigma_{c_2} = \emptyset$. We have, for this definition of sq_2 ad $\sigma_{sel} = \sigma_c \backslash \sigma_{c1}$, that the above equivalence is true.

Connected queries only Unless otherwise specified, all queries and subqueries mentioned from this point onwards are connected.

A first possible definition of candidate subqueries we are interested in, considers very broadly any subquery of at least one query:

Definition 5 (Candidate subquery) *Given a set Q of pqueries $q_1, \ldots, q_n$, a query s is a candidate subquery of Q if there exists $q \in Q$ such that s is a subquery of q (Definition 2).*

However, this definition leads to considering too many subqueries. In particular, if s is a subquery of only one query $q \in Q$, materializing s will reduce the cost of evaluating q, but not the other queries in Q. Thus, we are more interested in subqueries shared by several queries in Q. Further, among such shared subqueries, we are interested in those that apply as many computation steps as possible. This is formalized as follows:

3.3 Most specific subqueries

In this subsection, we define the most specific subquery, which is the optimal sharing unit that our algorithm should produce as an output for a set of queries.

Definition 6 (Most specific subquery of a set of queries) *Let Q be a set of queries.*

1. The subqueries of Q, *denoted S_Q, are subqueries shared by all the queries in Q:*

$$\{s \mid \forall q \in Q, s \text{ is a subquery of } q\}$$

2. *A* most specific subquery (msq, in short) *of Q is a subquery $s_1 \in S_Q$ such that s_1 is a subquery of no other query in S_Q.*

Note that a set of queries may have several msqs. Further, one may think that an msq has the largest number of atoms among all queries in S_Q. But this is not always true, as shown by the following example.

Example 5 (MSQ) *Let $Q = \{q_1, q_2\}$ such that q_1 is:*

$$q_1(\bar{u}) \text{ :- } \quad n(u_1, u_2, u_3, \underline{1}), e(u_1, A, u_4), n(u_4, u_5, u_6, \underline{2}), e(u_4, B, u_5), n(u_5, u_6, u_7, \underline{3})$$
$$e(u_5, C, u_8), n(u_8, u_9, u_10, \underline{4}), e(u_8, X, u_{11}), n(u_{11}, u_{12}, u_{13}, \underline{5}),$$
$$e(u_{11}, I, u_{12}), n(u_{12}, u_{13}, u_{14}, \underline{6}), e(u_{12}, J, u_{15}), n(u_{15}, u_{16}, u_{17}, \underline{7})$$

for some $\bar{u}$ not detailed. For readability, we introduce the ***shorthand*** *below to denote q_1:*

$$\underline{1} - A - \underline{2} - B - \underline{3} - C - \underline{4} - X - \underline{5} - I - \underline{6} - J - \underline{7}$$

Observe that the shorthand notation hides some information about the query, such as: the node labels (if they are known; in the above, we assumed they are variables, but if some of them had been constants, the shorthand would not have reflected them), the exact head variables, etc. That is fine (we know that an actual pquery needs to specify these details). Further, consider a query q_2, shown directly in the shorthand notation:

$$\underline{1} - A - \underline{2} - B - \underline{3} - C - \underline{4} - Y - \underline{5} - I - \underline{6} - J - \underline{7}$$

such that the subqueries between $\underline{1}$ and $\underline{4}$ are identical, and similarly those between $\underline{5}$ and $\underline{7}$. The underlined numbers, as usual, are node equivalence Then, these subqueries are two msq's, none of which is a subquery of the other. They do not have the same number of atoms, they are not over the same atoms, and their predicate sets are not comparable, since they carry over different atom sets.
One could try to say $\underline{1} - A - \underline{2} - B - \underline{3} - C - \underline{4}$ is better, since it is over 3 node atoms. But depending on the data cardinalities, $\underline{5} - I - \underline{6} - J - \underline{7}$ may be more beneficial.

4 Proposed algorithms for MQO on a set of pqueries

We are given as input a set Q of pqueries $\{q_1, ..., q_n\}$. We propose two algorithms: the first algorithm described in Section 4.1 computes the msq of a pair of pqueries $\{q_i, q_j\}$. The second algorithm, still in its preliminary stages and proposed in Section 4.2, determines how each query q in Q should be rewritten and processed based on the results of Algorithm 1. Section 4.3 covers some preliminary ideas that we have for the cost model, and section 4.4 details the project's current progress.

In this section, we denote the output of Algorithm 1 as Algo1($\{q_1, q_2\}$).

4.1 Algorithm 1: Identifying the most specific subqueries of two pqueries

Input: A pair of queries in Q, $\{q_i, q_j\}$

Output: The msqs of $\{q_i, q_j\}$

We start by introducing an alignment among sequences of atoms, through the node equivalence classes:

Equivalence class sequence For a query q, let the equivalence class sequence of q, denoted $ecs(q)$, be the sequence of equivalence classes of its node atoms.

Example 6 (Equivalence class sequence) *Consider again the query q_1 from Example 5. Its ecs is:*

$$\underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}$$

Now we present **Algorithm 1**:

1. Let S_{q_1,q_2} be the set of most specific subqueries of $ecs(q_1)$ and $ecs(q_2)$. Each element in S_{q_1,q_2} is a connected subsequence of node equivalence class values in q_1 and in q_2. By definition of msq, no element in S_{q_1,q_2} is a subsequence of another.
2. Each element $e \in S_{q_1,q_2}$ can be mapped to a subquery s_1^e of q_1, and a subquery s_2^e of q_2. Each subquery s has exactly the same node atoms as e, and s has additionally the edge atoms connecting these nodes atoms, all the predicates on the variables of these node and edge atoms, and corresponding head

variables.
By construction, the queries s_1^e, s_2^e have the same number of node atoms and edge atoms, and the node equivalence class values in node atoms appearing at the same position (index) in s_1^e and s_2^e are equal. This allows us to establish a one-to-one correspondence between node atoms, thus also between edge atoms, and from these, a **one-to-one correspondence between attributes (variables or constants)** in s_1^e, s_2^e.

3. For each $e \in S_{q_1,q_2}$, based on s_1^e, s_2^e and the correspondence between their attributes, create a pquery s^e, together with the rewritings r_1, r_2 of s_1^e, s_2^e as follows:
   (a) s^e is a pquery mirroring the node and edge atoms of s_1^e, s_2^e. Initially, s^e has a different (fresh) variable in each attribute, except for the node equivalence classes (where it has the values that s_1^e, s_2^e share).
   (b) Initially, r_1 and r_2 are $True$, i.e s_1^e and s_2^e do not need to be rewritten.
   (c) From the first s^e atom to the last, and for each attribute in that atom, starting from the first:
      - If s_1^e, s_2^e both have a constant in that position, and it is the same constant, replace all occurrences of the corresponding s^e variable with that constant.
      - Otherwise, do not modify s^e (leave a variable there). If s_1^e (respectively, s_2^e) had a constant in this position, add a selection predicate enforcing this predicate to r_1 (respectively, to r_2).
   (d) After traversing all attributes in this way, r_1, r_2 show how to rewrite s_1^e, s_2^e through selections on s^e. The rewritings of q_1, q_2 are obtained by joining r_1, r_2 with all the atoms in q_1 (respectively, q_2) that were not in e.

Extended shorthand We extend the shorthand notation of connected pqueries to also *show the node label, when known* (that is, when it is given as a constant in the query). Instead of specifying, for each node atom, just its equivalence class ID, e.g., $\underline{1}$, $\underline{2}$, etc., we use a lowercase letter from the beginning of the alphabet, with the equivalence class ID as an index, e.g., $a_{\underline{1}}$ designates a node atom whose label is specified to be the constant a, and whose equivalence class ID is 1.

Illustration of Algorithm 1

We will apply algorithm 1 on the following pqueries that are inspired by those in Example 5, namely:

$$q_1 : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - C - l_{\underline{4}} - X - e_{\underline{5}} - I - f_{\underline{6}} - J - g_{\underline{7}}$$
$$q_2 : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - \alpha - d_{\underline{4}} - Y - k_{\underline{8}} - I - f_{\underline{6}} - J - g_{\underline{7}}$$

A constant label in a node is represented by a lower-case letter, whereas a constant label in an edge is represented by an upper-case letter. The equivalence class of each node is indicated as a subscript (in the form of a number). Variables are represented by the Greek alphabet (e.g. α).

We use u_j^i and v_j^i to denote the variables in the canonical body q_0 of both queries, written in the following form:

$$q_0(x) :\text{-}\ n(u_1, u_2, u_3, \underline{1}), e(u_1, u_4, u_5), n(u_5, u_6, u_7, \underline{2}), ..., e(u_{25}, u_{28}, u_{29}), n(u_{30}, u_{31}, u_{32}, \underline{7})$$

Step 0: Define $ecs(q_1)$ and $ecs(q_2)$

$$ecs(q_1) : \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{5}, \underline{6}, \underline{7}$$
$$ecs(q_2) : \underline{1}, \underline{2}, \underline{3}, \underline{4}, \underline{8}, \underline{6}, \underline{7}$$

Step 1: Calculate the set of most specific subqueries S_{q_1,q_2} There are two most specific subqueries:

$$e_1 : \underline{1}, \underline{2}, \underline{3}, \underline{4}$$
$$e_2 : \underline{6}, \underline{7}$$

From Step 2 onwards, we only consider e_1 for illustration purposes (in reality, we need to apply the same steps to all elements in S_{q_1,q_2}). For readability, we will from this point onwards refer to e_1 as, simply, e.

Step 2: There is a one-to-one correspondence between e and the nodes (and their associated predicates) in q_1 and q_2. Map e to s_1^e and s_2^e:

$$s_1^e : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - C - l_{\underline{4}}$$
$$s_2^e : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - \alpha - d_{\underline{4}}$$

Step 3: Construction of s^e and rewritings r_1^e and r_2^e of s_1^e and s_2^e

3.1: Initialization of s^e, r_1^e and r_2^e:

$$s^e : x_{\underline{1}} - y_1 - x_{\underline{2}} - y_2 - x_{\underline{3}} - y_3 - x_{\underline{4}}$$

where all atoms are variables.

$$r_1^e = r_2^e = True$$

3.2.1: Iteration for each attribute on the first s^e atom: s_1^e and s_2^e have the same constant label in their first atom, hence we update the label of the first atom of s^e:

$$s^e : a_{\underline{1}} - y_1 - x_{\underline{2}} - y_2 - x_{\underline{3}} - y_3 - x_{\underline{4}}$$

3.2.2: Iteration on the second s^e atom: Again, s_1^e and s_2^e have the same constant label in their second atom (first edge), hence we update the label of the second atom of s^e:

$$s^e : a_{\underline{1}} - A - x_{\underline{2}} - y_2 - x_{\underline{3}} - y_3 - x_{\underline{4}}$$

3.2.3 to 3.2.5 proceed in the same vein, so we skip to 3.2.6.

3.2.6: Iteration on sixth atom: s_1^e has a constant label in its sixth atom whereas and s_2^e does not, hence we leave the variable y_3 unchanged:

$$s^e : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - y_3 - x_{\underline{4}}$$

Since s_1^e has a constant label in its sixth atom, add a selection predicate enforcing the label to r_1^e:

$$r_1^e : y_3 = C$$
$$r_2^e : True$$

Final output of 3.2 for e:

$$s^e : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - y_3 - x_{\underline{4}}$$
$$r_1^e : (y_3 = C) \wedge (x_{\underline{4}} = l_{\underline{4}})$$
$$r_2^e : (x_{\underline{4}} = d_{\underline{4}})$$

Rewriting q_1, q_2 based on e:

$$\text{Original } q_1 : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - C - l_{\underline{4}} - X - e_{\underline{5}} - I - f_{\underline{6}} - J - g_{\underline{7}}$$
$$\text{Rewritten } q_1 : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - y_3 - x_{\underline{4}} - X - e_{\underline{5}} - I - f_{\underline{6}} - J - g_{\underline{7}} \text{ with } r_1^e$$
$$\text{Original } q_2 : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - \alpha - d_{\underline{4}} - Y - k_{\underline{8}} - I - f_{\underline{6}} - J - g_{\underline{7}}$$
$$\text{Rewritten } q_2 : a_{\underline{1}} - A - b_{\underline{2}} - B - c_{\underline{3}} - y_3 - x_{\underline{4}} - Y - k_{\underline{8}} - I - f_{\underline{6}} - J - g_{\underline{7}} \text{ with } r_2^e$$

4.2 Algorithm 2: Greedy MQO algorithm

We have come up with a first idea for the second algorithm that determines how each query is rewritten and computed, as follows:

1. Let S be the subqueries of at least 1 query from Q[1]
2. For each $s \in S$, estimate the cost of evaluating s.
3. For each $q \in Q$, find the best evaluation method of q based on S.
4. Repeat
   - Select the subquery $s \in S$ maximizing the difference between the sum of its cost savings for any query in S and its estimated cost. The cost function has yet to be defined (refer to Section 4.3).
     (a) If such a subquery exists and leads to a positive difference:
        – Materialize s; remove s from S.
        – Each query whose best evaluation method used s will be evaluated in this way. Remove these queries from Q.
     (b) Else, q will be evaluated directly on the graph.

   until an evaluation method has been found for each query.

4.3 Quantitative measure of selectivity or cost

Here we include some preliminary thoughts for the evaluation of the selectivity of pqueries (i.e. the cost model) by considering the models used in the state-of-the-art.

[SSB+08] discussed the selectivity estimation for the conjunctive Basic Graph Pattern, in which they assume that for t = (s p o), the subject, predicate and object are statistically independent. In this case, the selectivity of t $sel(t) = sel(s) \times sel(p) \times sel(o)$ (refer to 5.1.5 for the definition of the selectivity of an edge).

A first idea would hence be to define the cost of evaluating a pquery $q(\bar{x}) :\!\text{-}\, an_1, ae_1, an_2, ae_2, \ldots, an_{k-1}, ae_{k-1}, an_k$ as $cost(q) = min(sel(o))$, where o ranges over all the node and edge atoms. The cost of an empty pquery is infinite. However, this idea does not give a cost that is precise enough for our purposes: the selectivity of each query would be determined by one of its atoms, and so each query will have the same selectivity as one of its component subqueries. This cost model is also not additive. Given a query q that can be decomposed into two subqueries s_1 and s_2, the sum of costs of s_1 and s_2 will not be representative of the true cost of computing q via these two subqueries.

[1] We may consider msq's of *at least 2 queries* (to enable factorization), or also of *one query* (to split large queries into smaller parts, which may make them easier to optimize by Postgres and thus easier to evaluate). We plan to decide this experimentally later on.

Once we have come up with a suitable cost model, we can use the following two definitions that define the cost of computing q based on a subquery s and find the best evaluation method of a query q:

Definition 7 (Cost of computing q based on the results of s) *Since s is a subquery of q, there exists a unique subquery t such that $q = \sigma(t \bowtie s)$. Given that q and s are connected, one of the following two holds:*

- *t is a connected pquery, or*
- *t is not connected, but there exist two connected pqueries t_1, t_2 such that $t = t_1 \times t_2$, and $q = \sigma(t_1 \bowtie s \bowtie t_2)$*

The cost of computing q, based on the results of s, is:

- *In the first case above: the cost of t;*
- *In the second case above: the sum of the costs of t_1 and t_2.*

Given a set S of subqueries of a given query q, we could evaluate q using one subquery from S, or two, or more, etc. For simplicity, **we will consider using at most one subquery for each query**[2].

Definition 8 (Best evaluation method of q) *Given a query q and assuming a set of subqueries S, the **best evaluation method of** q is:*

- *If S contains no subquery of q, then evaluating q directly on the graph.*
- *If S contains some subqueries of q:*
  - *Let min_{rew} be the rewriting of q of the form $q = \sigma(s \bowtie t)$ which has the least cost across all the subqueries s of q.*
  - *If min_{rew} < the cost of evaluating q directly, then the best evaluation method of q is $\sigma(s \bowtie t)$. In this case, we say that the **cost saving** of s is (cost of q - min_{rew})*
  - *Else, the best evaluation method is directly on the graph.*

4.4 Experiments in progress

1. Cost evaluation: using "explain" function of postgres
2. Actual cost of queries: determined by the time taken for

[2]This is because for each subquery s, we know how to rewrite q as "s join something". But we have not yet the knowledge of "how to rewrite q with s_1 **and** s_2".

5 Annexes

5.1 Annex 1: Preliminary notions

We include here a set of notions that have been discussed and written down when starting this research. They are mostly widely known in the scientific community; they are recalled here for reference.

5.1.1 Query languages and query dialects

A **query language** is characterized by (*i*) a **syntax** (what are the legal queries that one can write, i.e., what should the query language parser accept) and (*ii*) a **semantics**, that is: what does a given query mean on a given database on which it is asked, or, equivalently, what does the query return if evaluated on that database. The standard (definition) of a language typically defines both the syntax and the semantics. Here are a few examples:

- **SQL** is a query language for relational databases. Its syntax is standardized by the ISO committee (`https://www.iso.org/standard/63555.html`).
- **SPARQL** is a query language for RDF databases, standardized by the W3C (`https://www.w3.org/TR/rdf-sparql-query/`).
- The so-called **conjunctive queries (CQ)** for relational databases is not an industry standard, but it is closely derived from logic, and widely used in research, as it provides a simple and practical syntax.
  - A conjunctive query consists of a *head* and a *body*.
  - The body is a conjunction of *atoms*. Each atom refers to a table from the underlying database; the atom has as many positions as there are attributes in the table. Within the atom, some positions may be taken by variables, and some positions may be taken by constants.
  - The head is a subset of the variables of the body; this is what the query returns.

  For instance, the conjunctive query $q(x, y) \text{:-} R(x, y)$ returns all the (x, y) tuples from relation R. The CQ $q(x) \text{:-} R(x, 5)$ returns all the values of the first attribute in R from all the tuple where the second attribute is 5. The CQ $q(x, y) \text{:-} R(2, x), S(x, y)$ returns all combinations of x and y such that $(2, x) \in R$ and $(x, y) \in S$.

  CQs can only express *conjunctions*, thus the name: there has to be a tuple in R *and* a tuple in S such that...

  The semantics of CQ is usually defined considering that a relation is a *set* of tuples (set semantics, no duplicates).
- **CQs over graphs** CQs can be easily adapted to other data models than relational, by describing such a model as a set of relations. For instance, we can model ConnectionLens graphs using two relations: $nodes(id, type, label)$, $edge(idsource, idtarget, label)$. Then, the query

  $$q(x) \text{:-} \quad node(z, XML_NODE, 'book'), edge(z, u, \epsilon), node(u, XML_NODE, 'title'), \\ edge(u, v, \epsilon), node(v, XML_VALUE, x)$$

  returns all the book titles found in an XML document, i.e., if the database contains the nodes and edges obtained from:

  ⟨book⟩
  ⟨title⟩"Database theory and practice"⟨/title⟩
  ⟨/book⟩

  the query result is a single tuple whose single attribute is: "Database theory and practice".

  The above graph model, based on $nodes(id, type, label)$, $edge(idsource, idtarget, label)$, assumes that nodes have two attributes other than IDs, specifically, a type, and a label; this is the case in ConnectionLens graphs. In RDF, nodes have no IDs other than their labels, thus it suffices to use

$node(label), edge(lsource, ltarget, label)$, which can be further simplified into just $edge(source, target, label)$, or $triple(subject, property, object)$ (if we move the edge label from the third to the second position).

A **query dialect** is a subset of a query language, typically excluding some features of the full query language.

For instance, the **conjunctive dialect of SQL** contains unnested SQL queries, without aggregation, without group-by, and using only the keyword `and` in the `where` clause (not using the keywords `or`, `not` etc.)

If we consider that relations do not contain duplicates, and that every SQL query starts with `select distinct...`*, then CQs can express exactly what the conjunctive dialect of SQL can express.* This, and the fact that CQs are very concise (they do not require us to write the `select, from, where` keywords, not even `and`!) is a reason why *CQs are often a preferred formalisms for problemq related to queries.* As stated above, any result obtained for CQs immediately transfers back to conjunctive SQL.

5.1.2 Query semantics

The semantics of a query is typically defined using *embeddings*. We illustrate this below for the relational model.

Let $R_1, R_2, \ldots, R_n$ be a set of relations. Let $q(\bar{x}) \text{:-} a_1, a_2, \ldots, a_m$ be a relational CQ such that each a_i, $1 \leq i \leq m$ is an atom referring to a relation R_j, $1 \leq j, \leq n$. $\bar{x}$ is the tuple of variables in the head of the query. For instance, if $n = 3$ and the database consists of R_1, R_2, R_3, here are three simple queries:

query	$\bar{x}$	m	a_1	a_2	a_3
$q_1(x) \text{:-} R_1(5, x)$	(x)	1	$R_1(5, x)$		
$q_2(x) \text{:-} R_1(5, x), R_3(x, 7)$	(x)	2	$R_1(5, x)$	$R_3(x, 7)$	
$q_3(x, y) \text{:-} R_1(x, z), R_1(x, w), R_2(z, w, y)$	(x, y)	3	$R_1(x, z)$	$R_1(x, w)$	$R_2(z, w, y)$

Definition 9 (Embedding) *Given a database instance I (that is, for each relation $R_1, \ldots, R_n$, a set of tuples having the schema of that relation) and a CQ q, an* embedding *of q in I is a function mapping each atom $a_i \in q$, where a_i refers to the relation R_j, into an R_j tuple $\phi(a_i)$ present in the instance, such that*

1. *For each a_i, and each position in a_i, if a_i has a constant in that position, then $\phi(a_i)$ has the same constant in that position;*
2. *For each variable v that appears in the q atoms $\{a_v^1, \ldots, a_v^k\}$, for some $1 \leq k \leq m$, in positions $i_1, i_2, \ldots, i_k$, respectively, i_k, we have*

$$\phi(a_v^1)[i_1] = \phi(a_v^2)[i_2] = \ldots = \phi(a_v^k)[i_k]$$

where $\phi(a_v^1)[i_1]$ denotes the value of the tuple $\phi(a_v^1)$ at position i_1, and similarly for the others.

Example 7 *Consider $n = 3$ as above and the following instance, where we also assigned some identifiers t_i^j to tuples, to help refer to them in the text:*

I:

R_1	
t_1^1	$(5, 2)$
t_1^2	$(5, 3)$
t_1^3	$(3, 6)$

R_2	
t_2^1	$(3, 2, 9)$
t_2^2	$(1, 2, 3)$
t_2^3	$(6, 6, 2)$

R_3	
t_3^1	$(2, 7)$

- *Embeddings of q_1: there are two ways to embed q_1's single atom into a tuple in R_1 (two distinct embeddings), denoted ϕ_1^1 and ϕ_1^2:*
  - $\phi_1^1(a_1) = t_1^1$
  - $\phi_1^2(a_1) = t_1^2$
- *Embeddings of q_2: we need to embed a_1 in R_1 and a_2 in R_3, so that the first attribute of R_1 is 5, and the second attribute in the embedding of a_1 is equal to the first in the embedding of R_3. There is only one way:*

- $\phi_2^1(a_1) = t_1^1$, $\phi_2^1(a_2) = t_3^1$

- *Embeddings of q_3: by a similar reasoning, we obtain:*
  - $\phi_3^1(a_1) = t_1^2$, $\phi_3^1(a_2) = t_1^1$, $\phi_3^1(a_3) = t_2^1$
  - $\phi_3^2(a_1) = t_1^3$, $\phi_3^2(a_2) = t_1^3$, $\phi_3^2(a_3) = t_2^1$

Definition 10 (Query result (semantics)) *Let I be an instance of a database and q be a query whose head is $\bar{x}$. The result of q on I, denoted $q(I)$, is the set:*

$$\{\phi(\bar{x}) \mid \phi \text{ is an embedding of } q \text{ into } I\}$$

where $\phi(\bar{x})$ is the tuple of values from I, assigned by the embedding ϕ to the variables in the head of q.

Example 8 *Continuing with the sample queries q_1, q_2, q_3 and instance I above, we get:*

- $q_1(I) = \{(2), (3)\}$. *The first tuple is because of ϕ_1^1, while the second is because of ϕ_1^2.*
- $q_2(I) = \{(2)\}$, *due to the single embedding ϕ_2^2*
- $q_3(I) = \{(5, 9), (3, 2)\}$. *The first tuple is because of ϕ_3^1 (note that $\phi_3^1(a_1)[1] = \phi_3^1(a_2)[1] = \phi_3^1(x) = 5$, and $\phi_3^1(y) = 9$), while the second is because of ϕ_3^2 ($\phi_3^2(x) = 3, \phi_3^2(y) = 2$).*

5.1.3 Relational algebra vs. conjunctive queries

It is easy to see that relational algebra operators apply quite naturally on a semantic query, and produce another semantic query, as follows.

Definition 11 (Projection of a CQ) *Given a CQ q whose head is $\bar{x}$, and a projection π_{col} such that each variable in col appears in $\bar{x}$, the projection $\pi_{col}(q)$ is a CQ having the same body as q, and as head, only those variables in $\bar{x}$ that also appear in col.*

Definition 12 (Predicate and condition) *A predicate is an expression of the form $v = c$ where v is a variable and c is a constant, or of the form $v_1 = v_2$, where v_1, v_2 are variables. A condition is a conjunction of predicates. For simplicity, we assume that a variable appearing in a predicate of the form $v_1 = v_2$ does not appear in a predicate of the form $v_1 = c$. (It is easy to modify a condition that does not respect this constraint, into a condition that does respect it. For instance, $(v_1 = 5) \wedge (v_1 = v_2) \wedge (v_3 = v_4)$ can be equivalently rewritten as $(v_1 = 5) \wedge (v_2 = 5) \wedge (v_3 = v_4)$).*

Definition 13 (Selection of a CQ) *Given a CQ q and a condition cond over the variables of q, the selection $\sigma_{cond}(q)$ is CQ obtained from q modified as follows:*

- *For each predicate of the form $v_1 = v_2$ in cond (where v_1, v_2 are distinct variables), replace all occurrences of v_2 by v_1 (v_1, v_2 are both variables, and they play symmetric roles)*
- *For each predicate of the form $v = c$ in cond, replace all occurrences of v by c.*

Definition 14 (Join of two CQs) *Given two CQs q_1, q_2 whose heads are $\bar{x_1}, \bar{x_2}$, and such that some variables appear both in the body of q_1 and in the body of q_2, the natural join $q_1 \bowtie q_2$ is a CQ whose body contains all the atoms of q_1 and all the atoms of q_2, and whose head contains all the variables in $\bar{x_1}$, followed by all the variables in $\bar{x_2} \setminus \bar{x_1}$.*

Example 9 *Consider the query q_3 from Example 7.*

- $\pi_y(q_3)$ *is the CQ:* $q_3'(y) \text{ :- } R_1(x, z), R_1(x, w), R_2(z, w, y)$
- $\sigma_{(x=4)\wedge(y=z)}(q_3)$ *is the CQ:* $q_3''(4, y) \text{ :- } R_1(4, z), R_1(4, w), R_2(y, w, y)$
- *Now, consider also the CQ* $q_4(y, t) \text{ :- } R_2(y, t, 5)$. *Note that y is the common variable between q_3 and q_4. Then, $q_3 \bowtie q_4$ is the CQ:*

$$q_{3,4}(x, y, t) \text{ :- } R_1(x, z), R_1(x, w), R_2(z, w, y), R_2(y, t, 5)$$

5.1.4 Query containment and equivalence

Definition 15 (Query Containment) *We say query q is contained in q', denoted $q \subseteq q'$, iff for any database instance D, $q(D) \subseteq q'(D)$.*

- "find all the red boats" is contained in "find all the boats" (and also in: "find all the red boats").

Definition 16 (Query Equivalence) *We say q is equivalent to q', denoted $q \equiv q'$, if for any database instance D, $q(D) = q'(D)$ (or, equivalently, $q \subseteq q'$ and $q' \subseteq q$.*

Definition 17 (Query homomorphism) [3] *A homomorphism ϕ is a function that maps the atoms[4] from query q_1 into the atoms of query q_2 such that all the predicates that hold on some variables in q_1 also hold on their images in the variables of q_2.*

For instance, consider the queries:

- q_2: `select * from R r, S s where r.a=s.b and r.a=5;`
- q_1: `select * from R x1, S y1 where y1.b=5 and x1.a=y1.b and y1.c=7;`

In this case, there exists a homomorphism from q_2 to q_1: $\phi(r) = x_1$, $\phi(s) = y_1$. There does not exist a homomorphism in the other direction, because the condition y1.c=7 cannot be mapped into q_2.

Query equivalence vs. query homomorphism (Chandra and Merlin, 1977)

The first result of Chandra and Merlin is that:

> Given conjunctive queries q_1, q_2, we have $q_1 \subseteq q_2$ if and only if there exists an homomorphism from q_2 to q_1.

As a consequence (also in Chandra and Merlin):

> Given conjunctive queries q_1, q_2, we have $q_1 \equiv q_2$ if and only if there exists an homomorphism from q_2 to q_1 and one in the opposite direction, which is equivalent to saying that there exists an isomorphism between q_1 and q_2.

5.1.5 Selectivity

Selectivity in relational databases In the simplest, relational database setting, the selectivity of a selection is the ratio between the size of the selection output and the size of the selection input. Specifically, let e be a relational algebra expression and $\sigma_c(e)$ be a selection on top of e. Then, the selectivity of σ_c is: $|\sigma_c(e)|/|e|$.

For instance, if R is a relation of 1000 tuples, and $\sigma_{R.a=5}(R)$ returns only 1 tuple, then the selectivity of $\sigma_{R.a=5}$ is 0.001.

Selectivity also generalizes to *joins*. Recall that a join is a selection over a cartesian product: $R \bowtie_c S \equiv \sigma_c(R \times S)$. In such a context, the selectivity of $\bowtie_c$ is defined as: $|R \bowtie_c S|/|R \times S|$.

Selectivity in graph databases In this context, the definitions are not as clear-cut, but we can still provide good intuitions on what this could mean.

- In an RDF database, edges may have different labels, and some edge labels may be more popular than others. The selectivity of a triple pattern of the form $(?x, p, ?y)$ (all the edges labeled p) can be defined for instance as: the fraction of all the graph edges that are labeled p.

[3] Definition details depend on the chosen query language.

[4] What *atom* means depends on the query language. In the relational examples below, an atom is an occurrence of a relation, for instance, both q_1 and q_2 have two atoms, one referring to the R table, one referring to the S table.

- More generally, we can say a graph pattern query is selective if it has "few results" (depending on what we compare).
- Further, if we consider the two queries $q_1 = (?x, p, ?y)$ and $q_2(?x, p, ?y), (?y, q, ?z)$:
  - q_1 may have less results than q_2, if few targets of p edges have q outgoing edges; for instance, suppose only 1 out of 10 nodes that are targets of p are also sources of q, and each of these has only one outgoing edge labeled q.
  - q_1 may have more results that q_2, if (i) a good part of the targets of p are sources of q, and/or (ii) each of these has many q edges.

5.2 Annex 2: Details of Scalable Multi-Query Optimization for SPARQL

This annex contains a detailed summary of the algorithms proposed in the paper by Le et. al [LKDL12].

5.2.1 Bootstrapping

For each i from 1 to n, we construct its predicate set which is a set of the predicates asked for in the query. For instance, the predicate set of query Q_1 is : $Pred_1 = \{P_1, P_2, P_3, P_4\}$ and the predicate set of query Q_2 is : $Pred_2 = \{P_1, P_2, P_3, P_4, P_5\}$.

We then calculate Jaccard similarity of the predicate sets, with the Jaccard similarity between two sets A, B defined as :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| + |A \cup B|}$$

The Jaccard similarity between $Pred_1$ and $Pred_2$ would be equal to

$$J(Pred_1, Pred_2) = \frac{|P_1, P_2, P_3, P_4|}{|P_1, P_2, P_3, P_4, P_5|} = \frac{4}{5}$$

The article then states that we run *k-means* on set Q to generate a set $M = \{M_1, ..., M_k\}$ query clusters.

5.2.2 Refining query clusters with hierarchical clustering

We now have a set $M = \{M_1, ..., M_k\}$ of query clusters. Within each query cluster M_i for i in {1,...,k}, we apply hierarchical clustering on its queries as follows:

1. Input: Query cluster M_i
2. Initialise a set $C = \{C_1, ..., C_{|M_i|}\}$ of $|M_i|$ clusters (each cluster in C has exactly one query).
3. Apply hierarchical clustering. At each iteration of the algorithm, we identify the pair of most structurally-similar clusters C_i and C_j that have not yet been tested.
4. We determine whether it is more cost-efficient to merge these two clusters to form a single Type 2 query or to keep them separate as two Type 1 queries. This is done by calculating the maximal common edge subgraphs (MCES), detailed in the next subsection.
5. Output: $C = \{C_1, ..., C_p\}$ with p $\leq |M_i|$

The algorithm terminates as there is a finite number of clusters.

5.2.3 Finding the Maximum Common Edge Subgraphs between two clusters

We consider the set of query graphs in $C_i \cup C_j$, that we denote as $\{Q_1, ..., Q_p\}$. We wish to find the structural overlaps for this set of queries, i.e. find the maximum common edge subgraph between them.

Construction of linegraphs

Whitney's Theorem states that every graph, different from the triangle K_3 or the claw $K_{1,3}$, is uniquely determined by its linegraph [Whi32]. This is why the MCCS (Maximum Common Connected Subgraph) problem between two unlabeled connected graphs can be solved as an MCCIS (MCC Induced Subgraph) problem between their linegraphs [VV08] (it is desirable to perform this reduction from MCCS to MCCIS as more research has been done on solving MCCIS)[5]. To construct a linegraph $L(G)$ from a graph G, we create a node in $L(G)$ for every edge in G and add an edge between two nodes in $L(G)$ if ther corresponding edges in G share a common node.

More specifically in this paper, the context of MQO raises new requirements for the linegraph construction. Besides the typical inversion of nodes and edges from the query graph, this transformation also assigns one of four labels l_0, l_1, l_2, l_3 to each edge in the line graph. This corresponds to the four possible joins between the subjects and objects of two triple patterns :

- l_0 : subject-subject
- l_1 : subject-object
- l_2 : object-subject
- l_3 : object-object

Fine-grained optimizations during the construction of linegraphs (Section 2.1)

Building Cartesian products directly on multiple query-linegraphs is computationally complex, hence the paper proposes a two-stage optimization that prunes the queries and ensures that only the most promising substructures would be transformed into linegraphs.

- Stage 1: remove predicates that are not common to all queries in $\{Q_1, ..., Q_p\}$.
- Stage 2: After contructing the linegraphs on the remaining predicates, execute a further reduction of the size of the linegraph by computing the Boolean intersections for the $m^-[e]$ and $m^+[e]$ for each linegraph vertex e.

Building Cartesian products on the linegraphs (Section 2.2)

Definition: In this paper, the Cartesian product G $\times$ H of graphs G and H is a graph such that:

- the vertex set of G $\times$ H is the Cartesian product V(G) $\times$ V(H), with the additional requirement that the each vertex in the pair (u,v) in G $\times$ H must also have the same predicate P.
- two vertices (u,u') and (v,v') are adjacent in G G $\times$ H if and only if either u is adjacent to v in G and u' is adjacent to v' in H (strong connections), or u is not adjacent to v in G and u' is not adjacent to v (weak connections).

We recursively build product linegraphs for all the queries in $\{Q_1, ..., Q_p\}$. The construction theoretically causes the size of the product graph to increase exponentially, but empirically the heuristics in the two-stage optimization allow us to avoid this situation.

Clique detection in product graphs (Section 2.3)

A maximal clique (clique that cannot be enlarged) with a strong covering tree (i.e. tree with only strong connections) in the product graph equals to a MCES, or a maximal common sub-query.

[5]MCES is not mentioned in [VV08]; instead, they mention the MCCES problem. We can safely assume that [LKDL12] just gave it a different name...

In this algorithm, we proceed as follows:

1. Find all maximal cliques in the product graph using one of the efficient algorithms that exist, for example the branch and bound algorithm proposed by Östergård [Ös02].
2. For each discovered maximal clique, identify its sub-cliques (= subquery) with the maximal strong covering trees.
3. Sort these subqueries by their size in descending order, loop through the sorted list from the beginning and stop at the first substructure that intersects S. This step optimizes cost of sub-queries by looking for substructures that are both large in size and correspond to selective common sub-queries.
4. Merge the queries q_i, q_j whose common sub-query is reflected in K and also merge their corresponding clusters into a new cluster.

References

[ABB+21] Angelos-Christos Anadiotis, Oana Balalau, Théo Bouganim, Francesco Chimienti, Helena Galhardas, Mhd Yamen Haddad, Stéphane Horel, Ioana Manolescu, and Youssr Youssef. Empowering Investigative Journalism with Graph-based Heterogeneous Data Management. *Bulletin of the Technical Committee on Data Engineering*, September 2021.

[AMM22] Angelos Christos Anadiotis, Ioana Manolescu, and Madhulika Mohanty. Integrating Connection Search in Graph Queries. In *BDA 2022 - 38ème Conférence sur la Gestion de Données - Principes, Technologies et Applications*, Clermont-Ferrand, France, October 2022.

[BMU22] Nelly Barret, Ioana Manolescu, and Prajna Upadhyay. Abstra: Toward Generic Abstractions for Data of Any Model. In *CIKM 2022 - 31st ACM International Conference on Information and Knowledge Management*, Atlanta, Georgia / Hybrid, United States, October 2022.

[CDG+18] Camille Chanial, Rédouane Dziri, Helena Galhardas, Julien Leblay, Minh-Huong Le Nguyen, and Ioana Manolescu. ConnectionLens: Finding Connections Across Heterogeneous Data Sources. *Proceedings of the VLDB Endowment (PVLDB)*, 11:4, 2018.

[LKDL12] Wangchao Le, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li. Scalable multi-query optimization for sparql. In *2012 IEEE 28th International Conference on Data Engineering*, pages 666–677, 2012.

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, mar 1988.

[SSB+08] M. Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.

[SSN94] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data & Knowledge Engineering*, 12(2):197–222, 1994.

[VV08] Philippe Vismara and Benoît Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In Hoai An Le Thi, Pascal Bouvry, and Tao Pham Dinh, editors, *Modelling, Computation and Optimization in Information Systems and Management Sciences*, pages 358–368, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Whi32] Hassler Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54:61–79, 1932.

[Ös02] Patric R.J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1):197–207, 2002. Special Issue devoted to the 6th Twente Workshop on Graphs and Combinatorial Optimization.