On Efficient Processing of BPMN-Q Queries

Ahmed Awad\textsuperscript{a}, Sherif Sakr\textsuperscript{b}

\textsuperscript{a}Faculty of Computers and Information, Cairo University, Egypt
\textsuperscript{b}NICTA and University of New South Wales, Australia

Abstract

Business processes are central to the operation of both public and private organizations. With the rapid growth in the number of process models developed by different process designers, it becomes crucial for business process designers to be able to look up the repository for models that could handle a similar situation before developing new ones. Therefore, providing support for querying these business process repositories is a crucial requirement. BPMN-Q is a visual language that has been designed to query repositories of business process models.

In this paper, we present a novel approach for efficient evaluation of BPMN-Q queries. The approach is based on indexing process models by the transitive closure of their control flow relation as well as path indexes. The closure index is precomputed while the path index is incrementally built through processing of queries. These indexes are utilized to achieve an effective filtering process and an efficient verification check for BPMN-Q queries. The results of our experimental evaluation show the effectiveness of our proposed approach.

\textit{Keywords:} Business Process Repository, BPMN-Q, Querying, Indexing

1. Introduction

Business process modeling is an essential first phase in the business process engineering chain. Models are created by business analysts with an objective to capture business requirements, enable a better understanding of business processes, facilitate communication between business analysts and
IT experts, identify process improvement options and serve as a basis for executable business processes. Designing a new process model is a highly complex, time consuming and error prone task. As the number of business process models increases, providing business process designers with a capability of querying and reusing previously designed process models is of great practical value. Reusing implies the need for querying a process repository in order to find a suitable previous work that can be the basis for a new design. Moreover, the querying is on the content of the process model rather than a sort of keyword matching on meta data associated to process models. Additionally, with the huge size of process repositories, it is of utmost importance to have efficient querying techniques. Having a query technique (language) that takes a considerable amount of time to process a query would only encourage the user, e.g., the process designer, to give up looking for something to reuse from the repository and instead start from scratch designing a new process.

In a previous work, we introduced BPMN-Q [2] as a visual language to query repositories of business process models. The language has been applied in different scenarios where reuse or retrieval of process models based on a query pattern was needed [7, 22, 18]. In general the query processing of BPMN-Q matches a query graph to the process graph and identifies the sub-graph(s) that represent a match to the query, if there are any. Due to its richness in expressing queries against process models, the evaluation process of a BPMN-Q query can be quite expensive one. Thus, it is of crucial importance to develop an efficient query processor for BPMN-Q queries especially with the continuous growth in the sizes of process model repositories.

In principle, the evaluation process of a BPMN-Q query can be intuitively mapped to a graph matching problem where a BPMN-Q query is represented as a query graph against a process model repository (e.g. Oryx [11]) representing a graph database. Owing to the very expensive cost of pair-wise comparison of graph data structures, most of the proposed graph query processing techniques (e.g. [14, 23, 30]) rely on a strategy which consists of two main steps: filtering and verification. For a given graph database \( D \) and a graph query \( q \), the main objective of the filtering and verification methodology is to avoid comparing the query graph \( q \) with each graph database member \( g_i \in D \) to check whether \( g_i \) satisfies the conditions of \( q \). Hence, the processing of these querying techniques goes through the following three main steps:

1. **Off-line index construction**: generates a feature set \( F \) from the graph
database $D$. An inverted index is then built to map each feature to its associated list of related graphs.

2. **Filtering**: probes the constructed index of (step 1) to eliminate, as much as possible, members that are identified as being of the false results. The indexed features are used to produce a set of candidate graphs which is related to the feature of the query $q$.

3. **Verification**: checks each candidate graph $g$ from the filtering phase to verify whether it indeed satisfies the conditions of the query $q$.

Since the candidate answer set is in general smaller than the entire graph database, query processing using the indexes is significantly more efficient than the sequential scan approach. However, candidate verification is still very expensive since the size of the candidate answer set is at least equal to that of the exact answer set in the optimal case but it is usually larger for most of the queries. Therefore, reducing the size of the candidate answer set by removing as much as possible of the false positive graphs is the main criteria to evaluate the effectiveness of the filtering techniques.

In principle, the query processing of BPMN-Q queries is more challenging than the traditional graph queries in the sense that the edges between the nodes of the BPMN-Q queries are not always simple or direct connections that can be evaluated using intuitive retrieval mechanisms. However, BPMN-Q query edges can represent more complex and recursive types of connections between the nodes of the business process graph models.

In this article, we describe a novel approach for efficient evaluation of BPMN-Q queries. Our approach is based on indexing process models by the transitive closure of their control flow relation as well as path indexes. The closure index is precomputed as a feature for all process models in the repository. The path index, however, is incrementally built based on the queries that are run against the repository. The path index was meant to be incrementally built to avoid the overhead of huge computation costs of unused paths in queries. Using these indexes, we show how we can achieve a more effective filtering process and a more efficient verification check for BPMN-Q queries. In particular, we summarize the main contributions of this article as follows:

- We present two efficient indexing mechanisms for business process model graphs. Namely, transitive closure index and incremental path index.
- We describe two algorithms for efficient processing of BPMN-Q queries in both of its two phases for the query evaluation process: the filtration phase and the verification phase. To the best of our knowledge, all rel-
evant graph indexing techniques are able to only utilize their indexing structures at the filtering phase.

- We present a new algorithm to evaluate path edges in a BPMN-Q query based on the available information of the transitive closure index.
- We present a novel mechanism for incrementally building the path index according to the queries that are posed against the process model repository. The main aim of this incremental building mechanism is to avoid the overhead of the huge computation costs of unnecessary paths that are rarely or never used in the querying workloads.
- We conduct an extensive set of experiments to evaluate the efficiency and performance of our new algorithms using a data set of 1256 real business processes using a set of 10 BPMN-Q queries with different complexities.

The remainder of this article is organized as follows. Section 3 describes BPMN-Q constructs as well as the query processing. The major contribution of the paper comes in Section 4 where we describe how to index process models based on their transitive closure and paths and how an efficient query evaluation can be achieved using these indexes. The evaluation of our approach is presented in Section 5. Related work is discussed in Section 2 showing the limitations of the current graph indexing approaches for BPMN-Q query processing before we conclude the article in Section 6.

2. Related Work

Process models represent the blueprint for subsequent executable processes. Designing new process models is a complex and time consuming task. This task could be accelerated if process modelers can lookup repositories of process models for similar situations to the one in hand. Moreover, a process models residing in the repository might be large in size. It might be difficult for the process modeler to manually traverse it and check whether it matches what is in his mind. There are different BPMN modeling tools such as: Intalio Designer\(^1\), the Eclipse STP BPMN Modeler\(^2\) and ITP-Commerce’s Process Modeler\(^3\). To the best of our knowledge, none of these tools are enabled with querying capabilities over a repository of previously defined business process

\(^1\)http://www.intalio.com/products/designer/
\(^2\)http://www.eclipse.org/stp/bpmn/model/index.php
\(^3\)http://www.itp-commerce.com/
models. Several authors have developed process query languages to query about the allowed executions of a BPM [12, 19, 28]. The main difference between those graphical languages and BPMN-Q is that BPMN-Q is used to formulate queries about the business process model itself (i.e. its graphical structure), not about the state space of its executions. This makes it possible to use BPMN-Q for searching for modeling problems without having to compute the state space of all possible executions. Beeri et al. [8, 9] have presented a query language for querying business processes called BP-QL. The query language is designed based on the BPEL standard and thus focuses on querying executable processes. Our work focuses on the reuse of higher level business knowledge. Francescomarino and Tonella [13] have presented another query language which is working on the graphical structure of a model called BPMN VQL. The main purpose of this language is to find crosscutting concerns in BPM. BPMN-Q is however more expressive than BPMN VQL. For instance, generic nodes, negative paths, and variable names in BPMN-Q have no equivalent constructs in BPMN VQL. Recently, Jin et al. [17] proposed a traditional path-based approach for indexing process models which are represented using Petri Nets. In this approach, the supported query features are quite limited and primitive. In addition, it suffers from the traditional path-based indexing for graph databases [25].

There are several techniques that have been proposed to deal with the subgraph query processing problem. GraphGrep [14] is a path-based approach for processing graph queries. It enumerates all paths through each graph in a database until a maximum length \( L \) and records the number of occurrences of each path. Jin et al. [16] have applied this approach for indexing business process models. However, the main problem of this approach is that many false positive graphs could be returned in the filtering phase. In addition, the number of the indexed paths could drastically increase with the size and number of process model graphs in the repository and thus the index may not all fit in the main memory. Some researchers have focused on indexing and querying graph data using data mining techniques such as: GIndex [29], TreePi [31] and Tree+Δ [33]. In these approaches data mining methods are firstly applied to extract the frequent subgraphs (features) and identify the graphs in the database which contain those subgraphs. Clearly, the effectiveness of these approaches depends on the quality of the selected features. GString [15] is another graph indexing scheme which focus on decomposing chemical compounds into basic structures that have semantic meaning in the context of organic chemistry. In this approach the graph search problem
is converted into a string matching problem. We believe that converting sub-graph search queries into string matching problem could be an inefficient approach especially if the size of the graph database or the sub-graph query is large. Additionally, it is not trivial to extend the GString approach to support the processing of graph queries in other domain of applications. In practice, none of these techniques fits in the context of processing BPMN-Q queries. In particular, the query processing of BPMN-Q queries goes beyond the traditional sub-graph query processing in two main aspects:

1. BPMN-Q subgraph queries do not treat all nodes of the graph repository or graph query in the same way. Each node has its own type and characteristics. Moreover, the nodes of BPMN-Q queries can be generic nodes, which means they can be matched to nodes with different types on the source repository models.

2. The connections (edges) between the nodes of the subgraph query is not always simple or direct connections that can be evaluated using the intuitive retrieval mechanisms. However, these query edges can represent more complex and recursive types of connections (paths, negative paths and negative connections) between the nodes of the business process graph models.

3. The verification phase of these approaches does not fit with the 1-M nature of graph matching in BPMN-Q.

These limitations have motivated our approach, which we present in the rest of this article, for building an efficient BPMN-Q query processor that fills the existing gap in the graph indexing techniques for process models.

3. BPMN-Q: A Visual Language for Querying Business Processes

BPMN-Q [2, 26] is a visual language which is based on the concrete syntax of BPMN (Business Process Modeling Notation) [1], i.e., the visual notation. However, the language is designed to query core concepts of business process models, as shown in Definition 1. Thus, it is possible to use a concrete syntax based on EPCs and then the language can be EPC-Q, etc.

BPMN-Q is used to query business process diagrams by matching a business process diagram to a query graph. BPMN-Q is designed to access repositories of process models and structurally match the query to process models. The answer of a BPMN-Q query is represented by a set of process sub-graphs where each member of this set represents a refinement of the query. In practice, BPMN-Q can be used to find semantically similar process models by
using information retrieval techniques. That is, to find process models which have labels which are semantically similar to the labels in the query [4]. Also, BPMN-Q can be used to check the compliance between a business process model and existing business rules [3] or to provide an automated maintenance of process variants using process views [22, 21].

We will informally describe the elements of BPMN-Q in the next subsection. Later on, we formalize the notion of process diagrams, BPMN-Q queries and the matching of a query to a process diagram.

3.1. BPMN-Q Elements

A BPMN-Q query is represented as a business process diagram using the BPMN constructs that are augmented with querying-specific constructs (whose graphical representations are shown in Fig. 1) which are described as follows:

(a) **Anonymous Activity Node**: refers to (unknown) activities in a process diagram. (In order to distinguish non-anonymous activity nodes from anonymous activity nodes, we will refer to non-anonymous nodes as named activities.)

(b) **Generic Node**: refers to an unknown node in a process. It could evaluate to any node type (i.e. activities, join gateways or split gateways).

(c) **Generic Split**: refers to any type of split gateways.

(d) **Generic Join**: refers to any type of join gateways.

(e) **Negative Sequence Flow Edge**: states that there is no edge from a node A to a node B.

(f) **Path Edge**: states that there must be a path from a node A to a node B.

(g) **Negative Path Edge**: states that there is no *acyclic* path from a node A to a node B. That is, by removing cycles in the process graph we are not able to reach node B from A.

![BPMN-Q Elements](image)

**Figure 1: BPMN-Q Elements.**

The result of matching a query to a business process diagram is given by a set of sub-graphs of the original business process diagram. An exemplary
BPMN model and two queries are shown in Fig. 2. When matching the process graph in Fig. 2(b) to the query in Fig. 2(a), the result of the query is the process sub-graph shown in Fig. 3.

Some nodes in the diagram have a number. These numbers are not part of the BPMN notation; they will be used as references to the nodes later in the article.
It is possible to exclude some graph elements from a path search by assigning names to elements or to a path and adding an \textit{exclude} property to a \textit{path} edge: For instance, in the query in Fig 2(c), the marking \textit{Exclude(purchase raw material)} at path edge from the XOR-split to an end event means that the path must not contain an activity with such a label. A negative path between two nodes in a query requires that there are no acyclic paths between the nodes in the matching process diagram.

3.2. Matching Queries to Processes

In this section we describe the verification of a query graph against a process graph. We leave the filtration to the next section. To process a BPMN-Q query, a query graph is matched to the structure of a business process diagram.

In order to explain the matching algorithm formally, we will first define a business process diagram in the modeling language BPMN as a directed typed graph.

In the following, let \( \Sigma \) be an alphabet and \( \Sigma^+ \) be the set of non-empty words over \( \Sigma \). The labels (words from the set \( \Sigma^+ \)) can be assigned to elements of process graphs and query graphs.

**Definition 1.** A business process diagram (or process graph) is a tuple \( G_P = (N, \text{type}, F, l) \) where

- \( N \) is a finite set of nodes
- \( \text{type} : N \rightarrow \{\text{Activity, StartEvent, EndEvent, AND, XOR, OR}\} \) is a function that assigns a type to each node,
- \( F \subseteq N \times N \) (the edges in the graph) represents the sequence flow relation between the nodes,
- \( l : A \rightarrow \Sigma^+ \) is a labeling function.

For each process graph we have at least one start event and at least one end event. The set of start events can be defined as \( SE = \{x : x \in N \land \text{type}(x) = \text{StartEvent}\} \). Similarly, \( EE \) is the set of end events.

As described in Section 3.1, the query language BPMN-Q provides additional elements.

**Definition 2.** A query graph used for querying a process graph \( G_P = (N, \text{type}, F, l) \) is a tuple \( G_Q = (N_Q, \text{type}_Q, S, P, l_Q, \text{isNegative}, \text{exclude}, \text{ref}) \) where
• \( N_Q \) is a finite set of nodes,
• \( \text{type}_Q : N_Q \rightarrow \{ \text{Named activity}, \text{Anonymous activity}, \text{AND}, \text{XOR}, \text{OR}, \text{Generic}, \text{Generic split}, \text{Generic join}, \text{StartEvent}, \text{EndEvent} \} \) is a function that assigns a type to each node,
• \( S \subseteq N_Q \times N_Q \) is the set of sequence flow edges,
• \( P \subseteq N_Q \times N_Q \) is the multi-set of path edges. By using a multi-set, we allow that two path edges \( e_1, e_2 \in N_Q \times N_Q \) with \( e_1 = e_2 = (\text{start}, \text{end}) \in N_Q \times N_Q \) are contained in \( P \) as long as it holds that \( \text{exclude}(e_1) \neq \text{exclude}(e_2) \) (see below).
• \( l_Q : N_Q \cup \{ p \in P : \text{isNegative}(p) = \text{false} \} \rightarrow \Sigma^+ \) is a partial function that assigns labels to nodes and path edges.
• \( \text{isNegative} : P \cup S \rightarrow \{ \text{true}, \text{false} \} \) is a function that determines whether sequence flow and path edges are marked as negative,
• \( \text{exclude} : \{ p : p \in P \land \text{isNegative}(p) = \text{false} \} \rightarrow 2^{\Sigma^+} \) is a function that determines the set of node labels that have to be excluded from a match to a path edge.
• \( \text{ref} : N_Q \rightarrow N \cup \{ \text{unresolved} \} \) is a function that assigns nodes of the process graph \( G_P \) or the special element “unresolved” to the nodes of the query graph. If \( n_1, n_2 \in N_Q \) with \( n_1 \neq n_2 \) and \( \text{ref}(n_1), \text{ref}(n_2) \in N \), then \( \text{ref}(n_1) \neq \text{ref}(n_2) \). Initially, \( \text{ref}(n) = \text{unresolved} \) for all \( n \in N_Q \).

As can be seen from the definition above, nodes and non-negative paths of the query graph can be referenced by assigning labels. Labels can be activity names or special labels. Activity names can be used for nodes whose type is “named activity”. When finding a match to the query, such a node has to be matched by a node in \( G_P \) with exactly the same label. Special labels can be used for all kinds of nodes and for path edges. Sequence flow edges are not labeled. Special labels starts with ‘@’ (denoting anonymous activities) or with ‘?’ (denoting variables for other nodes and path edges). They are used to refer to specific nodes or path edges in \( \text{exclude} \) properties of other path edges. If a node label is used in the \( \text{exclude} \) property of a path edge, this means that this path edge refers to a sub-graph which does not contain nodes with such labels. A path edge that has the name of another path edge \( p_{ex} \) in its \( \text{exclude} \) property refers to a sub-graph which does not contain any node that is contained in the sub-graph matched by \( p_{ex} \). Of course, there must be no cyclic references among path edges.

A BPMN-Q query is matched to a business process graph via a set of
refinements to the query. With each refinement, nodes/edges in a query graph are replaced with nodes/edges of the matching process graph. We call such a replacement a resolution of an element of the query. If one node can have more than one possible replacement within the process model, a new refined copy of the query graph is created for each possible replacement. The nodes are resolved in a certain order, based on their types (e.g., named activity, gateway, generic node, etc.) so that each resolution step reduces the varieties of resolving other nodes. After each refinement step, the references to special labels are updated if their corresponding nodes or path edges were resolved by the query processor.

The order of resolution steps is as follows:
1. Resolve named activity nodes,
2. Resolve event nodes,
3. Resolve gateway nodes,
4. Resolve anonymous activity nodes,
5. Resolve generic splits and joins,
6. Resolve generic nodes,
7. Check unresolved sequence flow edges,
8. Check negative sequence flow edges and negative path edges,

Constraints that result from sequence flow edges in the query are taken into account while resolving the nodes.

The query processor starts its work with an unrefined query. This is a query graph whose nodes are all unresolved. With steps 1-6 above, nodes of the query graph are resolved by nodes of the process graph (formally, the function \( ref \) assigns process graph nodes to query graph nodes). For each resolved node, all references to that node in the query are updated to refer to the resolvent node of the process graph. In step 7, the query processor makes sure that all sequence flow edges in the query have counterparts in the matching process. Conversely, in step 8 the query processor makes sure that there are no sequence flow or path edges that are required to be absent. Finally, in step 9, path edges are substituted with nodes and sequence flow edges from the process graph. At the end, the query graph will be transformed into one or more matching sub-graph(s) of the original process model.

The query processor looks for all matches for a query within a process diagram. Thus, when a node in a query has more than one possible resolvent node in the process diagram, the query processor creates a new intermediate refinement copy for each possible resolution. Then, each query refinement is
examined further for matches for the rest of the unresolved nodes. This of course is of combinatorial nature and is expensive in processing, in Section 4, we discuss how to speedup this step.

For any intermediate refinement copy, if any of the steps above fails, this refinement is discarded. A query is said to find a match if at least one of the intermediate refinements evolves to a totally refined copy that can be mapped to a sub-graph of the process model.

To formally describe the query-process matching, we first describe the resolution of query nodes with process nodes.

A resolution of an activity node (steps 1 and 4) can be defined as follows:

**Definition 3.** [Resolution of activity nodes]

Let \( G_Q = (N_Q, \text{type}_Q, S, P, l_Q, \text{isNegative}, \text{exclude}, \text{ref}) \) be a query graph for querying the process graph \( G_P = (N_P, \text{type}_P, F, l_P) \).

The resolution of an activity node \( a \in N_Q \) with \( \text{type}(a) \in \{\text{Named activity}, \text{Anonymous activity}\} \) is a set \( Q \) of refinements. This set contains new query graphs in which the activity node \( a \) has been refined to an activity node \( a_P \in N_P \).

A query graph \( G_{\text{new}} = (N_Q, \text{type}_Q, S, P, l, C, \text{isNegative}, \text{exclude}, \text{ref}_{\text{new}}) \) is a refinement of the original query graph \( G_Q \) (i.e. \( G_{\text{new}} \in Q \)) iff

- If \( \text{type}(a) = \text{Name}\text{d activity} \), then \( l_Q(a) = l_P(a_P) \) (a named activity in the query graph is refined by an activity in the process graph which has a corresponding label),
- If \( \text{type}(a) = \text{Anonymous activity} \), then \( a_P \in N \) is selected from the activity nodes in the process graph such that the sequence flow constraints in \( S_Q \) are fulfilled, i.e., \( \forall x \in N_Q. \ \text{ref}_Q(x) \neq \text{unresolved} \wedge (x, a) \in S_Q \vee (a, x) \in S_Q \exists x' \in N. (a_P, x') \in F. \text{ref}(x) = x' \),
- \( \text{ref}_{\text{new}}(a) = a_P \),
- \( \text{ref}_{\text{new}}(x) = \text{ref}(x) \) for all \( x \neq a \).

Similarly, we can define resolution for the other node types. At the time where all nodes in an intermediate refinement query graph are resolved, this query graph is called a totally refined query.

**Definition 4.** [Totally refined query]

A query graph \( G_Q = (N_Q, \text{type}_Q, S, P, l_Q, C, \text{isNegative}, \text{exclude}, \text{ref}) \) is called a totally refined query iff \( \forall n \in N_Q : \text{ref}(n) \neq \text{unresolved} \).
When all nodes in the query graph have been refined, it remains to be checked whether the totally refined query adheres to the constraints given by the sequence flow and path edges. The check for (negative) sequence flows are straightforward steps. However, the substitution of path edges requires search in the process graph. This is performed such that a sub graph $\Gamma$ of the process graph is searched according to the following definition:

(To understand how resolution and substitution of path edges works, we start by describing reachability of nodes within a process graph.)

**Definition 5.** [Reachability: Node $m$ is reachable from node $n$]

Let $G_P = (N, type, F, l)$ be a process graph. Let $F^*$ be the transitive closure over a sequence flow relation $F$. For each node $n \in N$, we define the set of reachable nodes from $n$ as $[n]_{F^*} = \{m : (n, m) \in F^*\}$.

Based on the above definition, we can define the reachable nodes from some node $a$ under an additional “exclude” condition. For instance $[a]_{(F \setminus \{(x, y)\})^*}$ defines the set of reachable nodes from node $a$ in the case that the edge $(x, y)$ has been removed from the sequence flow relation $F$.

With the help of Definition 5, we can explain how path edges are processed by the query processor. A path edge $p = (source, target)$ can be resolved by a sub-graph $\Gamma$ of $G_P$ iff all nodes in $\Gamma$ are 1) reachable from source node 2) target node is reachable from every node in $\Gamma$ 3) $\Gamma$ does not contain any node $n$ whose label $l_P(n)$ is contained in exclude($p$) 4) $\Gamma$ does not contain any node that is reachable only from a node $n$ which label $l_P(n)$ is contained in exclude($p$) 5) edges between nodes in $\Gamma$ are those “between” these nodes in $G_P$.

**Definition 6.** [Resolving a path edge]

A path edge $p = (source, target)$ can be resolved by a sub-graph $\Gamma$ of the process graph $G_P = (N, type, F, l)$ if:

- For each node $n$ of $\Gamma$, there is a path from source to $n$ that does not include any node $x$ with $l(x) \in$ exclude($p$).
- For each node $n$ of $\Gamma$, there is a path from $n$ to target that does not include any node $x$ with $l(x) \in$ exclude($p$).
- $\Gamma$ is the “largest” subset with the above properties, i.e. there is no $\Gamma' \supset \Gamma$ whose nodes fulfills both above properties.

Checking a negative path $np = (source, target)$ is not simply to ensure that the operation to find a path that has the same source and destination would fail. Rather, negative paths are meant to check that acyclic
paths can not be found. This can be achieved by checking a path edge \( p = (\text{source}, \text{target}) \) where \( \text{exclude}(p) \) contains all nodes that causes cycles in the process graph in which \( \text{source} \) is involved yields an empty subgraph. We call these nodes loop entry nodes. To help illustrate the idea, consider the process graph in Figure 4.

![Figure 4: A Process model with a loop](image)

In that process model, a negative path \((C, F)\) does exist because the path from \(C\) to \(F\) happens only due to loops. That is, every path from \(C\) to \(F\) has to visit Join 1 which is the loop entry node. On the other hand, a negative path \((C, D)\) does not exist because \(D\) can be reached from \(C\) without visiting Join 1. Notice that in this specific process graph there is only one loop node. However, there could be more than one if the process contains more than one loop.

**Definition 7.** [Loop entry node] Let \( G = (N, \text{type}, F, l) \) be a process graph with \( SE \) being the set of its start events. A node \( n \in N \) is called a loop entry node iff

- It belongs to a loop: \((n, n) \in F^*\), and
- It is a join node: \(|\{x : (x, n) \in F\}| > 1\), and
- All reachable nodes from \(n\), except itself, will not be reachable from start nodes of the process in case outgoing edges of \(n\) were removed: \(\forall x \in [n]_F, \forall s \in SE, x \notin [s]_{(F \setminus \{(n, z) : (n, z) \in F\})^*}\).

After we have described the concepts in a formal way, we discuss the processing of the query in Figure 2(b) as an example. When the process graph in Figure 2(a) is matched to the query in Figure 2(b), first the named
activity node with label "Analyze order" is bound to the activity in the process with the same label. In the next step, the query processors looks for a match to the XOR-split node. Referring to the process in Figure 2(a), there are two XOR-split nodes. At this point of time, the query processor makes use of required sequence flow edges in the query, if any. In our specific case, the query processor selects the leftmost XOR-split (with ID 1) in the process as a resolvent to the respective node in the query, as this one has an incoming sequence flow edge from the named activity node "Analyze order" while the second XOR-split in the model has not. In the last step, the query processor tries to resolve the anonymous activity (the one with the label "@W"). Once more, the sequence flow connectivity in the query is used. Since the resolved XOR-split has an outgoing sequence flow edge to the named activity "Notify Customer", the anonymous activity is bound to this activity. The resulting match is shown in Figure 3.

When the query in Figure 2(c) has to be processed, the query processor starts resolving the event nodes in the query. Two refined copies of the query are created where in one copy an event node is bound to the end event 3 in the process and the other is bound to the end event 4, and vice versa in the other copy. To resolve the XOR-split, the query processor finds two resolvents, the two XOR-splits 1 and 2 in the process. There are no other sequence flow restrictions in the query that can help the processor to further eliminate one of the resolvents. Thus, four independent refinements are created as shown in Figure 5.

At this time all of them are totally refined. What is left to do is to check negative path edges and substitute path edges. The negative path checking will fail for the refinements in figures 5(a) and 5(b) because there is a path from the XOR-split 1 to end event 4 and from XOR-split 2 to end event 4 respectively in the process of Figure 2(a). Also, the check of the negative path edge in refinement 3 will fail because there is a path from the XOR-split 1 to end event 3 in the process. The check of negative paths will succeed for the fourth refinement as there is no chance to reach the end event 3 from XOR-split 2.

Only refinement 4 will proceed to the last step of substituting path edges. Finally, the query has only one result, the one shown in Figure 6.
4. An Efficient Query Processor

In general, it is clearly an inefficient and a very time-consuming approach to perform a sequential scan over the entire process model repository $R$ to check whether each process model $M$ belongs to the answer set of a BPMN-Q query $q$. Most of the proposed graph query processing techniques rely on indexing the graph repository using metrics and features from the set of graphs (e.g. Paths [14], Trees [31, 33], Sub-graphs [10, 15, 29]) which is then followed by the step of using the index information to, as much as possible, prune graphs from the candidate set of graphs during the query evaluation process. In principles, graph indices are generally useful to reduce the complexity of the query evaluation process from $O(N)$ to $O(logN)$ where $N$ is the number of process models in the underlying graph database [29, 23, 37] However, unfortunately, currently available sub-graph query processing techniques do not fulfill some of the main BPMN-Q query processing requirements (see Section 2 for detailed discussion) such as: the evaluation of path edges or the
1–M nature of BPMN-Q query matching, cf. Section 3.2. These limitations led us to develop our indexing technique that builds on existing approaches in order to fulfill the above-mentioned requirements gaps.

We use the (negative) path edge of BPMN-Q as the feature to be extracted and indexed. That is, within a process model, we need to know the nodes on a path from some node $A$ to another node $B$, cf. Definition 6. Moreover, we need to know these nodes in case that any other nodes are excluded from that path, the exclude property of a path edge, e.g., path from $A$ to $B$ excluding $C$. Current graph indexing approach either from the database community or the process management community fall short for the latter feature and thus is our contribution in this paper.

The path index is a hash table where the key is of the form $(n/p, source, target, EXC)$ where $n/p$ tells whether this entry was for the evaluation of (negative) path edge, the path edge $ep = (source, target)$ and $EXC = exclude(ep)$ determines the set of excluded nodes, this is always an empty set in the case of negative path edges. The entry value is a set of ordered pairs of the form $(mdl, M)$, where $mdl$ is the identifier of a process graph in which the match was found and $M$ is the set of nodes representing the matching subgraph within $mdl$, cf. Definition 6. However, building this index on process models is very expensive. For instance, for a process model with $n$ nodes, we need to evaluate $2^n * n^2$ path edge possibilities. $n^2$ to indicate that we need to compute path edges between all combinations of nodes; we need to repeat this $2^n$ times to reflect all possibilities of node exclusions. Moreover, each path edge evaluation uses a depth first approach to calculate the match. Obviously, this is a huge overhead. Let alone that some of these path combinations might never appear in any BPMN-Q query.

To overcome this dilemma of huge computation costs, our approach is based on a two-step indexing mechanism: the transitive closure index and
path index. The transitive closure index is pre-computed for the process graph’s control flow where the computation overhead is done once per process model per change. Nuutila [20] has an efficient algorithm for computing the transitive closure of relations. On the other hand, we build the path index incrementally during the processing of BPMN-Q queries. Thus, we create entries of the path index on demand.

4.1. Constructing Closure and Path Indexes

The ultimate objective to build the closure index is to help speed up path edges evaluation and thus to help incrementally build the path index, as discussed above. Yet, with the transitive closure computed, we can enhance BPMN-Q query processing in both the filtering and the verification phases. In the filtering phase, we use the transitive closure to check not only for direct edges between nodes but also for the existence of paths among them. This can be done in $O(1)$. More details about the enhancement of the filtering phase is discussed in Section 4.2. Also, in the verification phase, we can prune intermediate query refinements that are identified to be containing no matches to the query. Further details about this is described in Section 4.3. While the transitive closure index helps prune the set of candidate process models for a query on the form find paths from “A” to “F”, it does not contribute to the actual evaluation of the path edge, cf. Definition 6. That is, each time we evaluate the query find paths from “A” to “F” we have to recompute the subgraph containing the answer for each matching process model. What we are aiming at here is to compute this path once and then retrieve it in all subsequent evaluations of queries that contain a similar path edge. Moreover, the closure index does not help in the case of a query of the form find paths from “A” to “F” excluding “C” because with a direct lookup in the index we can decide whether “F” is reachable from “A” but the closure does not tell whether “C” is on the path or not. Additionally, filtration with negative path edges in the query is not always possible, recall the discussion in Section 3.2 about checking negative path edges. Indeed, checking for a negative path from $A$ to $X$ cannot be simply done by looking up the closure index and making sure that there is no entry for $(A, X)$ in the closure index, because there might be loops in the process model. These limitations of the closure index are handled by building the path index path index.

The path index is a hash table where the key is of the form $(n/p, source, target, EXC)$ and the entry value is a set of ordered pairs of $(mdl, M)$, as described above. The construction of the path index, i.e.,
putting entries in the hash table, depends on the queries issued against the process repository. Also, there is an overhead in calculating an index entry the first time a (negative) path edge is evaluated. Yet, rather than using a depth-first approach to evaluate the path edges, we depend on the closure index to find the matching sub-graph. Thus, the next time a query is processed (negative) path edges’ evaluation is always a constant time $O(1)$. Algorithm 1 shows how the closure index helps evaluate path edges that will be stored in the path index for a constant evaluation in later times.

Algorithm 1: Evaluation of a path edge using the closure index

**input**: $p$ (a process model), $pEdge$ (a path edge in a query), closure (the closure index)

**output**: $M$—The set of nodes constituting the matching sub-graph

$source^+ = \text{getReachableNodes} (pEdge.source, \text{closure});$

$target^+ = \text{getReachingNodes} (pEdge.target, \text{closure});$

if $pEdge.exclude \neq \emptyset$ then

$M = (source^+ \cap target^+) \setminus pEdge.exclude;$

if $\forall n \in M : (pEdge.source, n) \notin p.F$ then

return $\emptyset$—Connectivity is broken;

if $\forall n \in M : (n, pEdge.target) \notin p.F$ then

return $\emptyset$—Connectivity is broken;

toRemove = $\emptyset$;

forall the $n \in M$ do

if $\forall m \in M : (n, m) \notin p.F$ then

$\text{toRemove} = \text{toRemove} \cup \{n\}$—$n$ is not forward connected;

if $\forall m \in M : (m, n) \notin p.F$ then

$\text{toRemove} = \text{toRemove} \cup \{n\}$—$n$ is not backward connected;

$M = M \setminus \text{toRemove};$

return $M$;

Having a process model $p$, a path edge $pEdge$ and the closure index closure as input, Algorithm 1 computes the matching subgraph of $pEdge$ in $p$ as follows. First, the set of all nodes reachable from the source node of $pEdge$ is obtained by investigating the index. Similarly is the set of nodes reaching the target of $pEdge$. The intersection of the two sets constitutes the nodes in the matching subgraph $M$. Next, if the path edge has a non empty set of excluded nodes, they are removed from $M$. After that, connectivity is checked in the way that there is at least one node $n \in M$ where a control flow edge $(source, n)$ exists in $p$; at least one node $m \in M$ where a control
flow edge \((m, \text{target})\) exists in \(p\); each node \(v \in M\) must have at least one incoming and one outgoing control flow edge in \(p\). All nodes that do not obey the last condition are removed from \(M\). Finally, \(M\) is returned.

Figure 7 shows two sample process models and a snapshot of the closure and path index entries for these two processes. As mentioned earlier, the closure index is precomputed for all process models in the repository. An entry is described by a source and a target node and a list of process models in which target is reachable from source. cf. Figure 7(c). On the other hand, the path index is meant to be built incrementally while queries are evaluated, cf. Figure 7(d).

![Path models](image)

<table>
<thead>
<tr>
<th>Path</th>
<th>Process models</th>
</tr>
</thead>
<tbody>
<tr>
<td>(start, end)</td>
<td>PM1, PM2</td>
</tr>
<tr>
<td>(A, B)</td>
<td>PM1, PM2</td>
</tr>
<tr>
<td>(A, C)</td>
<td>PM1, PM2</td>
</tr>
<tr>
<td>(A, end)</td>
<td>PM1, PM2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(A, E)</td>
<td>PM1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>(B, D)</td>
<td>PM1</td>
</tr>
<tr>
<td>(A, F)</td>
<td>PM1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path/Negative path</th>
<th>(Process model, Matching subgraph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((p, \text{start}, \text{end}, \emptyset))</td>
<td>({(PM1, {\text{start}, A, B, Xor, Split, C, D, E, F, XorJoin, end})})</td>
</tr>
<tr>
<td>((n, A, E, \emptyset))</td>
<td>({(PM1, {A, B, XorSplit, D, E})})</td>
</tr>
<tr>
<td>((p, A, F, {C}))</td>
<td>({(PM1, {A, B, XorSplit, D, E, F})})</td>
</tr>
<tr>
<td>((p, A, C, {B}))</td>
<td>({(PM1, \emptyset), (PM2, \emptyset)})</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>((n, A, E, \emptyset))</td>
<td>({(PM2, \emptyset)})</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 7: Two process models with the closure and path indexes

In the following two subsections we will describe the details of the two
4.2. Filtering Phase

The query in Figure 8(a) consists of three activity nodes “A”, “B” and “E” with a sequence flow edge from “A” to “B” and a path edge from “A” to “E”. The processes in figures 7(a) and 7(b) represent two processes in the repository. The first step in answering these queries is to filter the graphs and prepare the set of candidate processes. Without the closure index, both processes will be included in the candidate set. That is because, both contain the three activities in the query and in both models there is a sequence flow edge from “A” to “B”. However, at verification time, the query processor will fail to find a path from “A” to “E” in the process in Figure 7(b).

On the other hand, with the availability of the closure index, cf. Figure 7(c), we can include path edges of the query in the filtration phase. That is, only the process in Figure 7(a) is a candidate, because we have an entry (A, E) in the closure index that points to PM1. What remains then at the verification phase is to evaluate the path edge to retrieve the sub-graph that contains both “A” and “E” according to Definition 6, if there was already no entry for it in the path index.
As said before, the closure index is good at determining whether a path exists between two nodes in the process graph. However, if the path edge in the query graph has its exclude property pointing to some other nodes, the closure index would not be of much help here. For instance there might be a path from node $A$ to node $B$. However, there might not be such path when we exclude node $C$.

**Algorithm 2:** Filtration of process graphs using indexes

**input:** $P$ (the set of all process models), $q$ (A BPMN-Q query),
$closure$ (the closure index), $path$ (the path index)

**output:** $C$—The set of candidate process models

$C = P$;

forall the $pEdge \in q.P$ do

if $\text{isNegative}(pEdge) = false \land \text{exclude}(pEdge) = \emptyset$ then

$R = \text{queryIndex}(\text{closure}, pEdge);$  
if $R \neq null$ then  
  $C = C \cap R;$

else if $\text{isNegative}(pEdge) = false \land \text{exclude}(pEdge) \neq \emptyset$ then

$R = \text{queryIndex}(\text{path}, (p, pEdge));$
if $R \neq null$ then  
  $C = C \cap \{pm : (pm, X) \in R \land X \neq \emptyset\};$
else if  

$\exists pTemp : pTemp.source = pEdge.source \land pTemp.target = pEdge.target \land pTemp.exclude \subset pEdge.exclude \land \text{queryIndex}(\text{path}, (p, pTemp)) \neq null$ then

$R = \text{queryIndex}(\text{path}, (p, pTemp));$
$C = C \setminus \{pm : (pm, X) \in R \land X = \emptyset\};$
else

$R = \text{queryIndex}(\text{closure}, (pTemp.source, pTemp.target, \emptyset));$
if $R \neq null$ then  
  $C = C \cap R;$

else if $\text{isNegative}(pEdge) = true$ then

$R = \text{queryIndex}(\text{path}, (n, pEdge));$
if $R \neq null$ then  
  $C = C \cap \{pm : (pm, X) \in R \land X = \emptyset\};$

return $C$;
To illustrate the role of the path index, let us consider the path index entry 

\[(p, A, F, \{C\}) = \{(PM1, \{A, B, XorSplit, D, E, F\})\}\]

of Figure 7(d), we know that in the process model PM1 there is a sub-graph that comprises all possible paths between the node A and F without visiting node C. At this point, we are sure that evaluation of the path edge will succeed and thus PM1 is included in the candidate set. Considering the query in Figure 8(c), using the path index, we can determine that neither PM1 nor PM2 can be included in the candidate set. That is because in the path index we have the entry 

\[(p, A, C, \{B\}) = \{(PM1, \{}\), (PM2, \{}\}\}.\]

Thus, when filtering the repository we know that the path edge from A to C excluding B will fail in both process models.

Also, the entries in the path index can help evaluate path edges other than the ones that directly correspond to entries in the index. Let us consider the query in Figure 8(d) where we have a path edge \((A, C)\) where 

\[\text{exclude}(A, C) = \{B, D\}\]

and this is the first time to evaluate such a path. However, in the index we have the entry 

\[(p, A, C, \{B\}) = \{(PM1, \{}\), (PM2, \{}\}\}.\]

That is, in the process graphs PM1 and PM2 there is no path from A to C without visiting B. At this point, we can conclude that the evaluation of the path edge \((A, C, \{B, D\})\) will fail because we require paths from A to C to exclude both B and D. Thus, PM1 and PM2 are also excluded from the candidate set. Moreover, the index is updated with the entry 

\[(p, A, C, \{B, D\}) = \{(PM1, \{}\), (PM2, \{}\}\}\]

directly without any overhead computations. On the other hand, if, for instance, we have an entry 

\[(p, A, C, \{B\}) = \{(PM1, \{}\), (PM2, \{}\), (PM3, \{A, M, C\}\}\}

in the index, i.e., in process graph PM3 there are paths from A to C excluding B, we have to include PM3 in the candidate set for the query in Figure 8(d) and at verification time we have to evaluate the path \((A, C), \text{exclude}(A, C) = \{B, D\}\) for PM3. We discuss the evaluation of path edges in Section 4.3.

Since a negative path edge on the form \(np = (\text{source, target})\) can be checked by checking a path edge on the form \(p = (\text{source, target, \{loop entry nodes of a process model PMi\}})\), we can benefit from the path index by including negative paths as well in the filtration. For preparing the candidate set of process models for the query in Figure 8(b), we find the entry 

\[(n, A, E, \{}\) = \{(PM2, \{}\}\}.\]

That is, in process model PM2 we have the negative path \((A, E)\) satisfied. Note that in PM2, Figure 7(b), there are no loop entry nodes. For process model PM1 this is not the case, there is actually a path from A to E. Thus, PM1 is excluded from the candidate set of process models to be verified against the query. Algorithm 2 summarizes
our filtration approach using the closure and path indexes.

4.3. Verification Phase

Once the set of candidate process models is defined, we need to verify each process model in the way described in Section 3.2. A node in the query graph can have more than one resolvent. That, of course, might cause an explosion of resolved query graphs of which some might end up having no matches. Using the closure and the path indexes, we can prune query resolutions that are with no matches earlier and thus reduce the processing time without affecting the final result of the query evaluation. To give an example, we refer to the query in Figure 2(c). The process in Figure 2(a) is a candidate for match to that query, we assign it the identifier $PM_{10}$. In addition to the closure index, assume that we have the following entries, among others, in the path index for that specific process.

\[
(n, \text{Xor Split 1}, \text{End event 4}, \{} ) = \{(PM_{10}, \{\text{Xor Split 1, Xor Join 1, \ldots, End event 4}\})\}
\]

\[
(n, \text{Xor Split 2}, \text{End event 4}, \{} ) = \{(PM_{10}, \{\text{Xor Split 2, And split 1, \ldots, End event 4}\})\}
\]

\[
(n, \text{Xor Split 1}, \text{End event 3}, \{} ) = \{(PM_{10}, \{\text{Xor Split 1, Notify customer, \ldots, End event 3}\})\}
\]

\[
(n, \text{Xor Split 2}, \text{End event 3}, \{} ) = \{(PM_{10}, \{}\})
\]

Now, the query processor starts with finding a match, as described in Section 3.2, by first resolving the end event nodes. To this end, there are two different resolutions where in each of the resolutions the two event nodes in the query are bound to the different event nodes in the process as shown in Figure 9.

Next, the query processor tries to resolve the XOR split. Without the help of the indexes, the query processor will try all possible combinations resulting in the final refinements shown in Figure 5. However, with the entries in the path index above, the processor can prune some refinements earlier. At this point, the XOR split in each refined query graph has two resolvents, XOR split 1 and XOR split 2 in the process model. Here, the query processor will check, using the path index, whether it should allow such a refinement. Beginning with XOR split 1 in the intermediate refinement of Figure 9(a), the processor checks the satisfaction of the negative path constraint between XOR split 1 and end event 3. Thus, the index path is queried with the key $(n, \text{XOR Split 1, End event 3, } \{} )$. This path exists in the index and has a non-empty result for $PM_{10}$ which is \{Xor Split 1, Notify customer, \ldots, End event 3\}. Thus, XOR split 1 is discarded from further evaluations. Now the query processor turns to XOR
split 2 and does the same check for the negative path edge to end event 3. The index has an entry for that path edge that is of empty result for PM10. Now, the query processor proceeds to check the path edge from XOR split 2 to end event 4 with the exclude property set to "Purchase raw material". That is, the index is queried with the key \((p, \text{XOR Split 2}, \text{End event 4}, \{ \text{Purchase raw material} \})\). For this path edge, the index has no entry. Thus, the query processor checks the weaker form of the edge against the closure index, i.e., \((p, \text{XOR Split 2}, \text{End event 4}, \{\})\). To this end, the query processor finds that there is a path from XOR split 2 to end event 4. The query processor generates a final refinement of the query as shown in Figure 5(d). The refinement in Figure 5(c) was pruned with the help of the path index. For the query refinement of Figure 5(d), all nodes have been refined and thus the query processor goes on and evaluates the path edge \((\text{XOR Split 2}, \text{End event 4}, \{ \text{Purchase raw material} \})\). The index path is updated with the result of the evaluation to be used in future queries. At this point the query has been evaluated and the result is reported to the user as in Figure 6.

Concerning the other intermediate refinement in Figure 9(b), the processor goes in the same way with the two candidate XOR splits. The check for the negative path between both XOR split 1 and XOR split 2 on the one hand and end event 4 on the other hand fails. Thus, this intermediate refinement is terminated. So, with the help of the path index, we have improved
the verification phase. In this specific example we have pruned 75% of the intermediate refinements of the query.

Algorithm 3 summarizes the approach described above to reduce possible resolvents of a node based on closure and path indexes. The algorithm shows the case where the resolved node $n$ is the source of a (negative) path edge. The case where $n$ is the target can be realized by repeating the If block where the check that $n$ is the source of the path edge and changing the condition to check that $n$ is the target of the edge and swapping the respective checks within the block.
Algorithm 3: Find node resolvents using the closure and path index

| input: n (node to be resolved), IR (Candidate resolvents), p (the process model to be verified), q (the query containing n), closure (the closure index), path (the path index) |
| output: RR—The reduced set of resolvents |

toRemove = ∅;

forall the r ∈ IR do
  forall the pEdge ∈ q.P do
    if n = pEdge.source and pEdge.target is resolved then
      Temp = queryIndex (path, (iif(isNegative(pEdge), n, p), r, pEdge.target, pEdge.exclude), p);
      if Temp ≠ null then
        — There is an index entry for that path edge;
        if ¬isNegative(pEdge) and Temp = ∅ then
          toRemove = toRemove ∪ {r};
        else if isNegative(pEdge) and Temp ≠ ∅ then
          toRemove = toRemove ∪ {r};
      else
        Temp = evaluatePathEdge(p, (r, pEdge.target, pEdge.exclude), closure) — based on Algorithm 1;
        updatePathIndex (iif(isNegative(pEdge), n, p), r, pEdge.target, pEdge.exclude, (p, Temp));
        if ¬isNegative(pEdge) and Temp = ∅ then
          toRemove = toRemove ∪ {r};
        else if isNegative(pEdge) and Temp ≠ ∅ then
          toRemove = toRemove ∪ {r};

RR = IR \ toRemove;
return RR;

4.4. Index Update

In any database, each update operation should be reflected to the related indices. In general, graph databases can be classified into two main categories [25]. The first category represents graph databases which consists of a large number of small graphs such as bioinformatic applications, chem-informatic applications and repositories of business process models [24]. The second category of graph databases usually represented as one (or very small
number) of large graphs such as social networks [32]. In practice, the problem of maintaining graph indices is much more challenging in the second category of graph databases because it is very hard to localize the effect of any graph edit operation (e.g. insert node/edge, remove node/edge). Hence, a large number of the entries in the associated index might need to be updated as a result. Our scenario belongs to the first category of graph databases where the problem of index maintenance is much easier due to the natural locality of any update operation. In particular, the effect of any update operation is, by default, limited to the entries of the associated graph. In addition, the average degree (number of outgoing edges) for the nodes of process model graphs is relatively very small. Therefore, the total number of paths which any node can belong to is mostly small as well.

In our context, when a model is updated, we recompute the whole associated entry of the model in the closure index (the old version is deleted and the new version is added) due to the very low cost of that operation. When the number of edit operations in the model is less than a specified threshold, we check which entries in the path index can be affected by this update operation and recompute them. If the number of edit operations exceed the defined limit, it becomes cheaper to delete the existing entries and perform a complete recomputation of the path index of the associated model. When a process model is deleted from the repository, all of its associated entries in our indices are removed as well.

5. Evaluation

We start this section by describing the architecture of BPMN-Q and its implementation. Next, we evaluate our approach by conducting a comprehensive set of experiments for querying business process models. The experiments of this section have the following goals:

1. Demonstrating the expressive power of our framework in dealing with different use cases.
2. Demonstrating the efficiency of our query engine and the effectiveness of the various optimization techniques.

5.1. Architecture

Figure 10 describes the overall architecture of BPMN-Q query engine which is built in a distributed client-server architecture. At the client level, the user is provided with the Query Designer component through which the
user can visually build a BPMN-Q query. The Query Interface component is responsible for shipping the query to the backend and then receiving the results. These results are presented to the user using the Result Presenter component.

At the server level, the Query Processor component receives the input query from the client side and perform all the querying matching steps as explained in section 3 and 4. The Process Model Indexer component is responsible for maintaining and accessing the indexes during the filtration and verification phases. In addition, it updates the index in case of changes in the underlying process repositories. The communication with external repositories takes place via the Uniform Language Interface whose responsibility is to parse the different serialization formats of the input process models and store them in a uniform representation, conforming to Definition 1, which is then accessible to the indexer and the query processor.

The architecture of our BPMN-Q query engine is realized within the Oryx business process model editor and repository [11]. In particular, the query processor is implemented in Java. The input process models are stored in
an Oryx-specific json format and the indexes are stored in flat files which are uploaded in the main memory once the query processor component is instantiated.

5.2. Performance Evaluation

In this section we report on the experimental evaluation of our proposed indexing and evaluation strategy for BPMN-Q queries. Initially, we have considered comparing our approach with some of the related approaches in querying repositories of business process models. However, it was not always to achieve an apple-to-apple comparison with similar approaches due to the following reasons:

- The BP-QL query language [8] is mainly designed for querying BPEL business processes. Therefore, it would be required to convert the whole experimental process model repository to BPEL format as a first step for comparing with BP-QL query processor. However, the main challenge is that the specification of the query language and use cases are different from the ones of BPMN-Q. Therefore, it can not support the expression and the execution of the semantics of most of our experimental queries (Figure 11). In addition, the implementation of the BP-QL query processor is not publicly available.

- The recently introduced BeehiveZ query processor [35] is another option that we considered for possible comparison. The implementation of the query processor is publicly available\(^5\). However, it is mainly designed for querying process models which are represented by Petri nets. The query engine only supports subgraph matching queries. Therefore it can not support the evaluation of most of the experimental queries (Figure 6).

- The VisTrails [36] is another publicly available project\(^6\). However, it is mainly focusing on querying workflows where only subgraph matching queries is supported. In addition, it is more-oriented for provenance use cases and is not suitable for our context.

In practice, it is not adequate to compare between different approaches without the same base of datasets, implementations and experimental queries [34]. Due to absence of such suitable apple-to-apple reasonable comparison with related work, we were unable to conduct such comparison. It is indeed a

\(^5\)http://code.google.com/p/beehivez/. However, it is mainly designed for querying process models which are represented by Petri nets. The query engine only supports subgraph matching queries. Therefore it can not support the evaluation of most of the experimental queries (Figure

\(^6\)http://www.vistrails.org/index.php/Main_Page

30
Figure 11: Queries used in the experiments

topic for future work to develop a uniform codebase for the different business process indexing techniques in order to compare their performance.
We have focused in our experimental evaluation on comparing the performance of the BPMN-Q query processing using filtration based on activity labels as well as direct sequence flow edges on the one hand. We show the performance of query processing with the availability of sequence flow transitive closure and also with the use of path index. In particular, we compare between the query processing times of four different settings:

1. Without the closure index [5].
2. With the closure index but without the path index.
3. With the closure index and the path index first run.
4. With the closure and the path index further runs.

The difference between the third and fourth settings is that the third setting includes the overhead of building the path index for the first time while the fourth setting represents further runs of the query where the index was already built. In our experiments, We have used a local installation of the Oryx process repository [11] with 1256 BPMN process models. These process models are from real business situations, for instance, they include the SAP reference models\(^7\). The labels “A”, “B”, etc are anonymizations for the actual labels of the queries in Figure 11. The total number of nodes in the process models collection is 21620 nodes. The total number of sequence flow edges is 20798. The time needed to build the transitive closure index was 405 seconds. The experiments were conducted on a MacBook Pro with 4 GB of memory and an Intel Core i5 processor with 2.4GHz processing speed.

We have tested 10 different queries with gradually increasing complexities and features that can be exploited to reduce the number of candidate process models. Figure 11 shows the 10 queries used in the experiments. Query 1, Figure 11(a) looks for two activities “A” and “F” and all nodes on path from the first to the last. Query 2 is a variant of query 1 where concrete activity “A” was replaced with an anonymous activity. Query 3 is a restriction of query 1 where we look for paths between “A” and “B” without visiting “C”. Query 4 puts more restriction by excluding “D” in addition to “C” from the path.

Queries 5-10 were developed in [18] with the purpose of identifying modeling errors, e.g., deadlocks in process models. That is, whenever a query finds a match within a process, the matched part constitutes a possible

\(^7\)The SAP reference models were transformed from EPC to BPMN using the approach in [27]
anomaly from the control flow perspective. Such queries are meant to be tested against the whole process model repository. Thus, it is not intended to reduce the candidate process models. However, as shown in Table 1 for some of the queries, using the closure index, the query processor is able to prune the search space. For further description of the queries, we refer the reader to [18].

Figure 12 shows the runtime of the 10 queries under the different indexes. It is obvious that the closure and/or the path indexing have advantages over the querying using only activity labels and sequence flow edges to prune the search space. Exceptions to this observation are the case of queries Q6 and Q7. In those two cases, the query processing based on the closure index performs worse than the without-index processor. Looking at those two cases, we find that looking up the closure index did not help prune the search space at the filtration phase, cf. Table 1. Moreover, at the verification phase, looking up the closure index did not help reduce the possible resolutions of the queries either. Thus, there is an overhead accessing the closure index without reducing search space. For all queries, the with the closure index and the path index first run query processor outperforms the two previous query processors although there is the overhead of constructing the index entries. This is due to the speed of Algorithm 1 in evaluating (negative) path edges over standard depth-first algorithms. This also gives an indicator about the performance of index updates by simply recomputing the index.
Figure 12: Runtime of the different query processors in milliseconds, queries 1–5 and 8 entries when a process model is modified.

For queries 5, 6, 7 and 8 the number of candidate process models is the same for all of the different query processors. That is, neither closure nor path indexes help prune the search space at the filtration phase. However, we notice the significant improvement of processing these queries using the path index. This is due to the pruning the search space at the verification
phase. Closely looking at one case, we find that the processing query 7 drops from 273.84 seconds without any indexing support to only 28.4 seconds using the path index. This is an evidence of the importance of pruning the search space for query resolutions at verification phase on the overall performance. For instance, for one of the investigated models, the number of resolutions without search space pruning is 6092 query resolutions. This number is reduced to 99 resolutions only using the path index. This shows how the verification phase can benefit as well from the index to prune the search space.

6. Conclusions

In this paper we have proposed indexing approaches for business process models for the sake of efficient processing of BPMN-Q queries. Two indexes were developed, the closure and path indexes. The two indexes helped to prune candidates at the filtration and verification phase of query processing. By filtering processes not just by node labels and sequence flow edges but also by path and negative path edges, a smaller set of candidates can be achieved. Even if it is not possible to eliminate processes models from the candidate set, because the query contains no path/negative path edges between activity nodes, pruning resolvents of query nodes at verification time is possible with the help of the indexes.

The closure index is precomputed over the set of process models in the repository. The path index is, on the other hand, built incrementally based on the closure index as well as the input queries to the processor. The reason to have two indexes is to provide a trade-off between the overhead of precomputing the path index and the efficiency of query processing. Our experiments show that even with the incremental building of the path index the query processor outperforms other query processors of BPMN-Q.

In this paper, we only discussed the query processing of control flow aspects of BPMN-Q queries. As a future work, we plan to extend the language to address other process models querying aspects, e.g., vertical navigation in sub-processes as well as data and role aspects. It is expected that these language extensions will call for adapting the query processor with further steps to filter and verify these aspects according to the input query. However, the work presented in this paper is not hindered by these expected extensions. Rather, we foresee the need of building further types of indexes to suit the nature of the extended querying aspects.


[32] Peixiang Zhao and Jiawei Han. On Graph Query Optimization in Large Networks. PVLDB, 3(1):340–351, 2010.


