Efficient Querying of Large Process Model Repositories

Tao Jin[a,b,1], Jianmin Wang[b], Marcello La Rosa[c], Arthur ter Hofstede[c,d,2], Lijie Wen[b]

[a]*Department of Computer Science and Technology, Tsinghua University, China*
[b]*School of Software, Tsinghua University, China*
[c]*Queensland University of Technology, Australia*
[d]*Eindhoven University of Technology, The Netherlands*

Abstract

Recent years have seen an increased uptake of business process management technology in industries. This has resulted in organizations trying to manage large collections of business process models. One of the challenges facing these organizations concerns the retrieval of models from large business process model repositories. For example, in some cases new process models may be derived from existing models, thus finding these models and adapting them may be more effective and less error-prone than developing them from scratch. Since process model repositories may be large, query evaluation may be time consuming. Hence, we investigate the use of indexes to speed up this evaluation process. To make our approach more applicable, we consider the semantic similarity between labels. Experiments are conducted to demonstrate that our approach is efficient.

Keywords: business process model, query, accurate, efficient, repository

1. Introduction

Through the application of Business Process Management (BPM) technology, organizations are in a position to rapidly build information systems

Email addresses: `jint05@mails.thu.edu.cn` (Tao Jin), `jimwang@tsinghua.edu.cn` (Jianmin Wang), `m.larosa@qut.edu.au` (Marcello La Rosa), `a.terhofstede@qut.edu.au` (Arthur ter Hofstede), `wenlj00@mails.thu.edu.cn` (Lijie Wen)

[1]Visit to QUT funded by China Scholarship Council.
[2]Senior Visiting Scholar of Tsinghua University.

and to evolve them due to environmental changes, e.g. in legislation or in market demand. BPM has matured in recent years and has seen significant uptake in a variety of industries, e.g. health, finance, manufacturing, and also in government. As a result, in many cases organizations have created large collections of business process models. For example, there are more than 6,000 models in Australia Suncorp Group, and there are more than 200,000 models in China CNR Corporation Limited. These models are often represented as graphs.

Managing large business process model repositories can be challenging. For example, when new process models are to be created one may wish to leverage existing process models in order to preserve best practice or simply to reuse process fragments. Therefore, one needs to have the ability to query a business process model repository. Due to the potential size of such a repository, it is important that these queries can be executed in an efficient manner.

In this paper focus is on the provision of efficient support for querying business process model repositories. Given a process fragment (the model query), we are concerned with finding all the process models in the repository that contain this fragment as subgraph. The complexity of finding all subgraph isomorphisms is known to be NP-complete [1]. To overcome this issue, and in line with graph database techniques [1, 2, 3, 4, 5, 6, 7], we propose a two-stage approach that reduces the number of models needed to be checked for subgraph isomorphism. Firstly, we filter the model repository through the use of indexes and obtain a set of candidate process models. Secondly, we apply an adaptation of Ullman's subgraph isomorphism check [8] in order to refine the set of candidate models, to extract those models containing the model query as subgraph. The advantage of using indexes is that the subgraph isomorphism check is only performed on a subset of the models in the repository, which is typically much smaller than the total number of models in the repository, so the query efficiency is improved.

The choice of which process model features to be indexed, and which logical data structure to be used to store indexes, is determined by the following requirements:

1. features should be efficiently extracted from a process model (i.e. a model query or a model in the repository);
2. indexes should be stored efficiently;
3. operations over indexes should be efficient (e.g. it should be possi-

ble to update the index incrementally as the process model repository changes);

4. it should be possible to use any fragment of a process model as a query (e.g. an isolated process node).

Accordingly, (i) we use task nodes as feature, as their extraction time is linear to the number of nodes in a model, their storage size is limited, and they can be used to look for isolated nodes, and (ii) we store our index in an inverted index, since this data structure allows efficient operations and can be updated incrementally.

In order to deal with different formats in a uniform way, we assume business processes be modeled as YAWL (Yet Another Workflow Language) models [9] or mapped from other formalisms into YAWL models. The expressiveness of YAWL is high and it offers comprehensive support for the control-flow patterns documented in [10]. In fact, it has been shown that a wide range of business process modeling languages used in practice, e.g. BPMN (Business Process Modeling Notation), EPCs (Event-driven Process Chain), UML ADs (Unified Modeling Language Activity Diagrams) and BPEL (Business Process Execution Language), or at least significant subsets of them, can be mapped to YAWL models directly or indirectly through Petri nets. For example, BPMN models can be transformed into YAWL models in [11], and BPEL models can be transformed into YAWL models in [12]. An alternative is that we can transform the other languages to Petri nets first and then transform Petri nets to YAWL models. You can see [13] for an overview of transformations from various BPM languages to Petri nets.

This work is an extended version of [14]. Our contribution can be summarized as follows.

- To improve query efficiency, we use an index storing the mapping from process tasks to process models where these tasks occur. This inverted index works as a filter, we only need to conduct the subgraph isomorphism check on the candidate models that pass this filter. The number of candidate models is always much smaller than the size of the repository, so the use of this filter can improve the query efficiency. Based on the findings reported in [14], we only build the index on the paths of length one here (as paths consist of task sequences, a path of length one consists of a single task).

- To increase the accuracy of query results, we consider data and resource aspects in addition to control-flow aspects when checking whether a model fragment is a subgraph of a candidate model. To deal with models with data and resource information, we work on YAWL models instead of Petri nets as in [14].

- To make the approach in [14] more applicable, we consider in this paper a notion of semantic similarity between labels. Tasks for which the degree of label similarity is greater than or equal to a given threshold are considered to be identical.
- We extend our implementation in the BeeHiveZ environment to account for the proposed extensions and conduct further experiments to evaluate the performance with data and resources.

The rest of this paper is organized as follows. Section 2 provides an introduction to YAWL and formally defines the semantics of a business process model query. Section 3 discusses the construction of indexes on the logical level while Section 4 shows how to query a business process model repository without label similarity. Section 5 shows how to deal with the semantic similarity between labels. Section 6 analyzes the use of our approach through a number of experiments on both a synthetic data set and a real data set. Section 7 discusses related work whilst Section 8 concludes this paper.

2. Preliminaries

The workflow language YAWL [9] is a general and powerful language grounded in the workflow patterns [10] and in Petri nets [15]. YAWL is the result of an in-depth analysis of control-flow constructs in workflows, and provides direct support for a number of patterns that are difficult to realize in Petri nets (e.g. cancellation, multiple instances). The definition of the syntax of YAWL can be found in [16], while the syntax and the semantics of the extension of YAWL, i.e. newYAWL, can be found in [17]. We use a simplified version of the definition of the syntax of YAWL net (YNet) as presented in [16]. This definition is presented below.

Definition 1 (YNet)**.** *A YNet is a tuple* $(C, T, F, Split, Join, V, VR, VW, R, RO, l)$ *where:*

- C *is a set of conditions,* $i \in C$ *is the input condition,* $o \in C$ *is the output condition;*

- *T is the set of tasks, for every $t \in T$;*
- *$F \subseteq ((C \setminus \{o\}) \times T) \cup (T \times (C \setminus \{i\})) \cup (T \times T)$ is the flow relation;*
- *$Split : T \nrightarrow \{AND, XOR, OR\}$ specifies the split behavior of each task;*
- *$Join : T \nrightarrow \{AND, XOR, OR\}$ specifies the join behavior of each task;*
- *V is the set of variables processed in the net, for every $v \in V$;*
- *$VR : T \nrightarrow \mathbb{P}(Var)$ specifies the variables read by every task;*
- *$VW : T \nrightarrow \mathbb{P}(Var)$ specifies the variables written by every task;*
- *R is the set of roles used in the net, for every $r \in R$;*
- *$RO : T \nrightarrow \mathbb{P}(Role)$ captures which roles are authorised to execute instances of which tasks;*
- *$l : T \cup V \cup R \to L$ is the label function where L is the set of labels.*

Example 1. *Figure 1 shows four example business process models represented as YNets, where only the control-flow perspective is shown. Data and resource perspectives are not graphically represented.*

In this paper, a query on a business process model repository is a YNet, and the result is defined as all YNets in the repository that cover that query. First we need to introduce the notion of a net covering another net.

Definition 2 (YNet Cover)**.** *YNet $yn_1 = (C_1, T_1, F_1, Split_1, Join_1, V_1, VR_1, VW_1, R_1, RO_1)$ is covered by YNet $yn_2 = (C_2, T_2, F_2, Split_2, Join_2, V_2, VR_2, VW_2, R_2, RO_2)$, denoted as $yn_1 \sqsubseteq yn_2$, iff there exists a one-to-one function $h : C_1 \to C_2 \cup T_1 \to T_2 \cup F_1 \to F_2$ such that:*

1. *for all $t \in T_1$: $l(t) = l(h(t)) \wedge Split(t) = Split(h(t)) \wedge Join(t) = Join(h(t)) \wedge VR(t) \subseteq VR(h(t)) \wedge VW(t) \subseteq VW(h(t)) \wedge RO(t) \subseteq RO(h(t))$, i.e. function h preserves task labels together with the data read and written and authorised roles;*
2. *for all $(n_1, n_2) \in F_1$: $h(n_1, n_2) = (h(n_1), h(n_2))$, i.e. h preserves arc relations.*

Definition 3 (YNet Query)**.** *Let R be a YNet model repository and let q be a YNet query. The result of issuing q over R is $R_q = \{r \in R \mid q \sqsubseteq r\}$.*

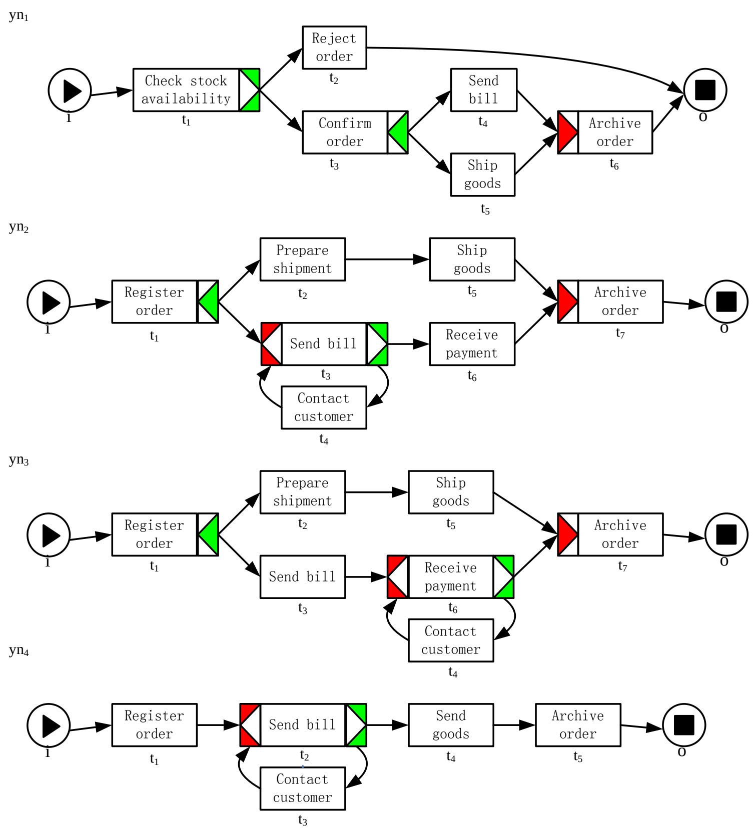


Figure 1: Business process model examples represented as YNets

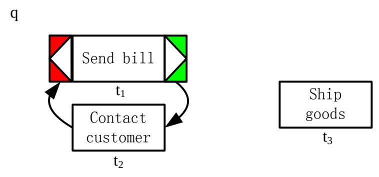


Figure 2: An example of a disconnected YNet used as a query

Example 2. *If we use YNet q in Figure 2 as a query and consider the four models in Figure 1 to constitute the Process model repository R, then only model yn_2 covers q (the semantic similarity between labels is not considered now), i.e. $R_q = \{yn_2\}$.*

3. Index Construction

To improve retrieval efficiency, and in-line with document retrieval systems [18], we use two inverted indexes, namely, the *task index* and the *label index*. We describe the *task index* in Section 3.1 and the *label index* in Section 3.2. The *task index* stores the mapping from process tasks to process models, and it can be used as a filter. The *label index* stores the mapping from words to task labels, and it can be used to retrieve similar labels.

3.1. Task Index

Based on the findings reported in [14], only paths of length one are indexed (as paths consist of task sequences, a path of length one consists of a single task). The mapping from tasks to models is stored in the *task index*. The *task index* takes the form of set of pairs ($task, model\ list$) where *task* denotes a task occurring in some models and *model list* denotes the set of these models where this task occurs. Given a model, we first extract all the tasks from this model. The extraction of tasks from a model takes linear time in terms of the number of tasks in the model. Then the mapping between the tasks and the models is set up in the *task index*.

Example 3. *Given the model repository shown in Figure 1, we can obtain all tasks. Some of these tasks are shown in Table 1. Then a* task index *can be created based on this data.*

Table 1: A sample of indexed items for tasks in Figure 1

task	model list
"Send bill"	yn_1, yn_2, yn_3, yn_4
"Ship goods"	yn_1, yn_2, yn_3
"Send goods"	yn_4
"Contact customer"	yn_2, yn_3, yn_4
⋮	⋮

3.2. Label Index

The *label index* is an inverted index that stores the mapping from words to the labels in which these words appear. The *label index* takes the form of set of pairs ($word, label\ list$) where *word* denotes the word appearing in some labels and *label list* denotes the set of these labels where this word appears. During the label index construction, we first extract all words from the labels and then convert them to lower case. Subsequently, we remove all stop words such as conjunctions and articles, and stem all remaining words (e.g. "transporting", "transported" and "transports" all become "transport"). When a new model is added to the repository, all the task labels can be extracted, and then the label index can be updated.

Example 4. *Given the model repository shown in Figure 1, we can obtain all labels. Some of these labels are shown in Table 2. Then a* label index *can be created based on this data.*

Table 2: A sample of indexed items for labels in Figure 1

word	label list
"Send"	l_1, l_3
"Ship"	l_2
"goods"	l_1, l_2
"bill"	l_3
⋮	⋮

label id	label
l_1	"Send goods"
l_2	"Ship goods"
l_3	"Send bill"
l_4	"Contact customer"
⋮	⋮

4. Query Processing without Label Similarity

YNet query processing is divided into two stages, namely, a filtering stage and a refinement stage. The first stage of query processing only acts as a filter. The reason is that we only use the task index in this stage, and context information is not taken into account, e.g. the fact that a choice may exist between two subsequent tasks in a model is lost as well as the order of the tasks with respect to the query. In order to further refine the set of process models so that it corresponds to an exact result, a match operation needs to be performed in the second stage.

The procedure for query processing is described in Algorithm 1. Line 1 extracts all tasks from the YNet acting as the query. Lines 2-5 work as a filter and a set of candidate models is obtained, which achieves the first stage of query processing. For each extracted task, Line 3 obtains the set of models containing that task using the *task index*. In Line 5 the intersection of the resulting candidate sets is computed so as to retain only those candidate models that contain all the extracted tasks. In Lines 6-8 each candidate model is checked in order to determine whether it exactly covers the query, thus implementing the second stage of query processing. In Line 7, the function CBMTest, explained later, is applied to take the context of these tasks into account.

Algorithm 1: YNetQuery

```
input  : q: the YNet query
output: a set of YNets covering q

// filtering stage
1 tasks ← taskExtraction(q);
2 foreach task in tasks do
3     list ← taskIndexQuery(task);
4     add list to lists;
5 R ← Intersection(lists);
// refinement stage
6 foreach c ∈ R do
7     if CBMTest(q, c) fails then
8         delete c from R;
9 return R;
```

Function CBMTest implements the YNet cover check according to Definition 2. It is an adaptation of Ullman's graph isomorphism algorithm [8] to YNets. In [19] an overview of graph matching algorithms is provided and it is stated that "Ullman's algorithm is widely known and, despite its age, it is still widely used and is probably the most popular graph matching algorithm". That is why in our approach Ullman's algorithm is chosen and adapted to YNets.

In CBMTest, task t can be mapped to task t' if and only if t and t' share the same label and the type of join (split) is same, the data read (written) by t is a subset of the data read (written) by t', and the set of roles linked to t is a subset of the roles linked to t'. Condition c can be mapped to condition c' if and only if there is a one-to-one mapping between the input tasks of c and c' and also between the output tasks of c and c'. CBMTest tries to find a one-to-one mapping between the nodes of the YNet query and the candidate YNet being investigated. All arcs in the YNet query must be preserved in the candidate YNet. If this is impossible, the candidate YNet is removed from the result set.

Example 5. *Consider q in Figure 2 as a YNet query. Consider yn_3 in Figure 1 as a candidate YNet. Tasks t_1, t_2 and t_3 in q can be mapped to t_3, t_4 and t_5 in yn_3 respectively. But the two arcs in q cannot be preserved in yn_3. Hence q cannot be covered by yn_3. On the other hand, consider yn_2 in Figure 1 as a candidate YNet. In that case, t_1, t_2 and t_3 in q can be mapped to t_3, t_4 and t_5 in yn_2 respectively, while the two arcs in q can be preserved. So yn_2 covers q.*

Example 6. *Consider the model repository $R = \{yn_1, yn_2, yn_3, yn_4\}$ as shown in Figure 1. We use YNet q in Figure 2 as a YNet query. First, we extract all tasks from q. This yields three tasks, t_1^q = "Send bill", t_2^q = "Contact customer" and t_3^q = "Ship goods". Then we can query the repository using the task index to obtain the models containing the corresponding tasks, $t_1^q.list = \{yn_1, yn_2, yn_3, yn_4\}$, $t_2^q.list = \{yn_2, yn_3, yn_4\}$ and $t_3^q.list=\{yn_1,yn_2,yn_3\}$. These sets correspond to those shown in Table 1. After taking the intersection of these sets, we get the candidate model set $R_q^c = \{yn_2, yn_3\}$. Now, the first stage of query processing is completed. To get the final result, we must determine which candidate models contain the model query as a subgraph. This yields $R_q = \{yn_2\}$.*

5. Dealing with Semantic Similarity between Labels

Business analysts may choose different names for identical tasks in process models. In order to recognize that different labels essentially represent the same task we consider a similarity notion for labels. Labels that have a degree of semantic similarity greater than or equal to a certain specified threshold are considered to be equal, hence two tasks having sufficiently similar labels are considered equal for the purpose of retrieval.

The following requirements are imposed on the use of a similarity notion for labels as part of the approach for querying:

1. it should be up to the user to decide whether or not to use the notion of label similarity during query process;
2. the user should be able to specify their preferred similarity threshold during query process;
3. the efficiency of query with label similarity should be as high as possible.

In order to fulfill these requirements, (i) we leave the construction of the task index unchanged. There is no dependence on whether the similarity notion is used or not, and therefore the task index does not need to be reconstructed; (ii) queries can take a similarity threshold into account. (iii) we construct a label index[3] to speed up the retrieval of similar labels and this index is not affected by the similarity threshold;

5.1. Semantic Similarity between Labels

Let $W(l)$ be the number of words that can be extracted from the label l and let $SCW(l_1, l_2)$ be the number of words in l_1 that appear in l_2 or have synonyms in l_2. The similarity between labels l_1 and l_2 can be calculated as Equation 1 (inspired by Dice's coefficient [20]):

$$sim(l_1, l_2) = \frac{2 \times SCW(l_1, l_2)}{W(l_1) + W(l_2)}. \quad (1)$$

Example 7. *Let l_1 = "Send goods" and l_2 = "Ship goods". These labels can be found in Table 2. From both labels we obtain two words. As the words "send" and "ship" are considered synonyms, we obtain the following degree of similarity: $sim(l_1, l_2) = \frac{2\times 2}{2+2} = 1$.*

Before the degree of similarity is computed, words are converted to lower case, stop words (e.g. "the" and "a") are removed and then words are stemmed. Naturally, Equation 1 can be replaced with another label similarity measure as long as this measure is based on the words used in the labels and not e.g. the characters.

5.2. Query Expansion

In order to take label similarity into account we need to make changes to the filtering stage and the refinement stage.

During the filtering stage, for every task in the query, we first obtain the task label, then obtain all labels in the repository that are similar to these labels. These retrieved labels are then used for retrieval purpose. Algorithm 1 is updated to Algorithm 2. Line 3 obtains all the similar labels according to a specific task. When we query on the label index, first, all words are

[3]Only task labels are stored, the labels of variables and roles are considered in the subgraph isomorphism check.

extracted from the task label used for querying and these words are treated the same way as during label index construction. Then these words are augmented with their synonyms and this extended collection of words is used for label retrieval. Labels that are sufficiently similar according to the specified threshold become a part of the output collection. After Lines 4-5 are finished, we obtain a list of models where the given task or the similar tasks occur. Here, we can see that the label similarity has effect on the query efficiency, because it affects the size of candidate models.

Algorithm 2: YNetQuery

```
input  : q: the YNet query
output: R: a set of YNets cover q
1  tasks ← taskExtraction(q);
2  foreach task in tasks do
3      similarTasks ← labelIndexQuery(task);
4      foreach similarTask in similarTasks do
5          list ← taskIndexQuery(similarTask);
6      add list to lists;
7  R ← Intersection(lists);
8  foreach c ∈ R do
9      if CBMTest(q, c) fails then
10         delete c from R;
11 return R;
```

During the refinement stage label similarity is used in Function `CBMTest` by considering tasks t and t' as equal if and only if the degree of similarity of their labels is greater than or equal to the specified threshold. We deal with the data names and role names in the same way when we check whether t and t' are a match.

Example 8. *Let us consider Example 6 again but now in the context of label similarity. Assume that the threshold for label similarity is set at* 0.9. *In that case the task t_3^q = "Ship goods" is expanded to the tasks* "Ship goods" *and* "Send goods". *t_3^q.list is updated to t_3^q.list = $\{yn_1, yn_2, yn_3, yn_4\}$, which gives us $R_q^c = \{yn_2, yn_3, yn_4\}$ as the set of candidate models. During the refinement stage, t_3 in q can be mapped to t_4 in yn_4 because the similarity* *of their labels is greater than the specified threshold, while t_1 and t_2 in q can be mapped to t_2 and t_3 in yn_4 respectively. Finally, apart from yn_2, yn_4 now also appears in the final result, that is, $R_q = \{yn_2, yn_4\}$.*

6. Tool Support and Evaluation

In order to evaluate our approach, we implemented it in our system named BeehiveZ[4]. BeehiveZ is a Java application, which makes use of the MySQL RDBMS to store process models as data type TEXT. The *task index* and the *label index* both are managed by Apache Lucene [21]. We decided to use Lucene as it is a search engine specifically designed for efficient text searches. Alternatively, we could have implemented the inverted indexes by using a two-column table on a RDBMS. The YAWL library was used for the representation of YNets. The ProM [22] library was used for display of YNets. To retrieve synonyms quickly, we stored a hash map of WordNet synonyms in main memory, occupying about 10MB. The use of label similarity and the specification of the threshold can be configured in BeehiveZ.

We conducted a number of experiments, both on a synthetic data set and on a real data set consisting of SAP Reference Models to determine the efficiency of our approach presented in the previous sections. To this end a computer with Intel(R) Core(TM)2 Duo CPU E8400 @3.00GHz and 3GB memory was used. This computer ran Windows XP Professional SP3 and JDK6, the memory of JVM was configured as 1GB.

6.1. Experiments on A Synthetic Data Set

All synthetic models were generated automatically using an algorithm that produces a collection of YNets randomly. The rules used in our generator come from [23]. Those rules were used for generating the control-flow perspective, we generate the variables that are read and written by tasks and the roles that are associated with tasks were generated in a random manner. Because all the labels in this data set are randomly generated character strings, we disabled label similarity during the query processing.

To evaluate the efficiency of our approach, we conducted some experiments. In these experiments 10 models were generated to act as queries and more than 600,000 models were generated to populate the business process

[4]BeehiveZ can be downloaded from http://sourceforge.net/projects/beehivez/

model repository. The 10 queries were evaluated each time after the addition of a certain number of freshly generated process models. Table 3 shows the characteristics of the various queries q_i, $1 \leq i \leq 10$. Specifically, for repository R, there were 600,210 YNets, the number of tasks in the various models ranged from 1 to 50[5], the number of conditions from 2 to 60, the number of arcs from 2 to 316, and there were at most 242,234 differently labeled tasks out of 15,605,435 tasks in total.

Table 3: Characteristics of model queries

	Number of tasks	Number of conditions	Number of arcs
q_1	1	2	2
q_2	6	8	16
q_3	11	9	24
q_4	16	13	35
q_5	21	25	60
q_6	26	28	71
q_7	31	24	68
q_8	36	36	106
q_9	41	31	130
q_{10}	46	39	116

We define R_q^c as the candidate set of answers resulting from q applied to the repository R using the index, while R_q^f denotes the final set of answers. Furthermore, we define T_s as the index traversal time, $T_{I/O}$ as the disk I/O time required to fetch each candidate YNet from disk, and T_v as the time required to compute whether there is an exact match.

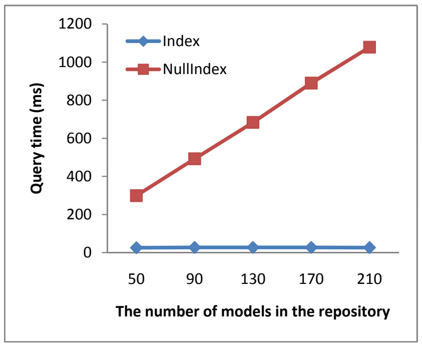
The query response time when an index is used can be computed using Equation 2.

$$T_q = T_s + |R_q^c| \times T_{I/O} + |R_q^c| \times T_v. \tag{2}$$

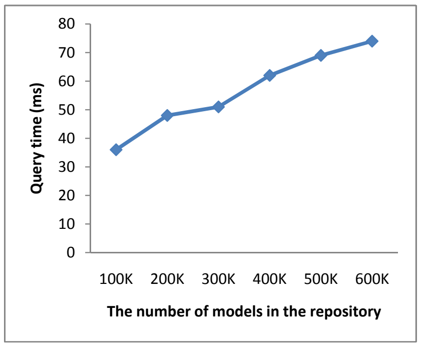
Equation 3 provides the query response time when no index is used.

$$T_q = |R| \times T_{I/O} + |R| \times T_v. \tag{3}$$

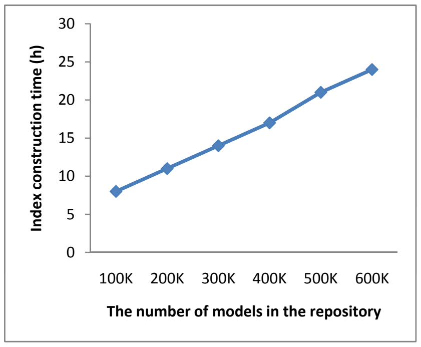
[5]According to 7PMG proposed in [24], models should be decomposed if they have more than 50 elements. So we generated models with the maximum number of tasks as 50, the number of conditions and arcs in a model is not configurable.



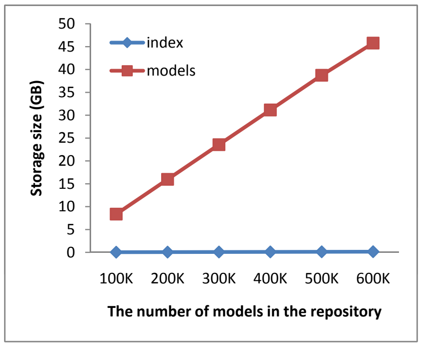
(a) query time comparison



(b) query time



(c) index construction time



(d) storage size

Figure 3: Performance of indexes

As $|R_q^c|$ is often much smaller than $|R|$, the use of the *task index* can save a significant amount of time, as shown in Figure 3(a). The more models there are in the repository, the greater the amount of time that can be saved through the use of our approach. Since for every query, the change of query time is similar, we use the average query time here.

Figure 3(b) shows the average query time when the *task index* is used. We can see that when more models are added to the repository, query evaluation becomes more time-consuming, though the time taken can still be considered acceptable.

Figure 3(c) shows the accumulated time required for index construction from scratch. Whenever a new model is added to the repository, the index does not need to be reconstructed but can simply be modified . Therefore the time required for index construction is acceptable. The storage size of indexes (including the *task index* and the *label index*) is shown in Figure 3(d).

The storage space for the indexes is about 1% of the storage space required for the models.

Now, we can draw a conclusion that our approach can improve the query efficiency significantly at little cost.

6.2. Experiments on SAP Reference Models

The SAP reference models are represented as EPCs and only contain control-flow information. They were first transformed into YNets using ProM[22][6]. This resulted in 591 YNets as 13 SAP reference models could not be mapped to YNets using ProM and these models were omitted from consideration. Some of the resulting models have more than one input conditions and/or have more than one output conditions in the model obtained by ProM. We merged these input (output) conditions into one in order to comply with the syntax of YAWL. Moreover, these models only contain control-flow elements. The number of tasks in the various models ranged from 1 to 53, the number of conditions from 2 to 66, the number of arcs from 2 to 145, and there were at most 1,494 differently labeled tasks out of 4,439 tasks in total. On this data set experiments were conducted with different similarity thresholds. First all models were added to the repository and the task index and the label index were built. Then every model was used as a query on the repository.

Table 4: The sizes of sets resulting from the use of different label similarity thresholds

	disabled	0.5	0.6	0.7	0.8	0.9	1.0
Min	1	1	1	1	1	1	1
Max	8	53	22	12	12	8	8
Average	1.58	2.73	2.02	1.75	1.71	1.62	1.62
St. Dev.	1.34	4.44	2.18	1.52	1.49	1.35	1.35

From Table 4 we can see that the size of the resulting sets increases when the label similarity is enabled, and decreases when the similarity threshold is higher. The query times also changed with the different label similarity thresholds. The results can be found in Table 5.

[6]We first transformed the EPCs to Petri nets, and then transformed the Petri nets to YAWL models.

Table 5: Query time (ms) with different label similarity thresholds

	disabled	0.5	0.6	0.7	0.8	0.9	1.0
Min	0*	0*	0*	0*	0*	0*	0*
Max	94	1981	1944	1203	1194	1194	1190
Average	5.87	74.65	63.14	55.99	55.25	53.12	52.99
St. Dev.	10.51	133.37	128.32	90.57	89.97	88.56	88.36

* It cost less than 1 millisecond.

From Tables 4 and 5, we can come to the conclusion that with the introduction of label similarity more models tend to be found, and that the sizes of the resulting sets of models decrease when the label similarity threshold increases. The query times change the same way, but they are still acceptable.

7. Related Work

Our work is inspired by the *filtering and verification* approach used in graph indexing algorithms. Given a graph database and a graph query, these algorithms are used to improve the efficiency of finding all graphs in the database that contain the graph query as a subgraph, by discarding the models that do not have to be checked for subgraph isomorphism. Different graph indexing algorithms use different graph features as indexes. For example, the GraphGrep [1] is an index on paths, the gIndex [2] is an index on frequent and discriminative subgraphs, the TreePi [3] is an index on frequent subtrees, and the FG-Index [4] is an index on subgraphs. In [6] frequent tree structures and a small number of discriminative subgraphs are used as indexing features. In [7], a summary of sub-structures is first built and then an index is constructed on them, while in [5], all connected induced subgraphs stored in the database are first enumerated and then organized in an index. In [25], a Closure-tree index is built on subgraphs. All these approaches work on abstract graphs with one type of node, while our approach operates on YAWL models which has two types of nodes. Additionally, in [2, 3, 4, 6] focus is on frequent sub-structures and thus they cannot efficiently deal with queries consisting of isolated nodes or infrequent sub-structures, and the major limitation of the Closure-tree index [25] is the high cost of the filtering phase due to the expensive structure comparison and the maximal matching algorithm. Hence, these approaches are not suitable for querying process model repositories, where each task may have a different label (thus reducing the frequency of common substructures) and queries may be made up of isolated tasks only. Moreover, extraction of sub-structures is more time-consuming and the storage space required increases. Another common drawback of these graph indexing algorithms is that the index is constructed using statistics on frequent features which are usually computed off-line, thus indexes cannot be easily updated when a new model is inserted into the graph database. Our approach on the other hand allows us to update the current indexes whenever a new model is added to the repository.

Our work has also commonalities with query languages for process models. For example, the Business Process Query Language (BPQL) [26] is a query language integrated with a process definition language, which allows one to query a process model or its running cases for the purpose of specifying flexibility requirements. BPMN-Q [27] is a visual query language which can be used for querying a repository of BPMN models. The query itself is expressed as a BPMN model where wildcard nodes and arcs can be used to articulate the query. Another visual query language is BPMN VQL [28], whose objective is to allow designers to identify, document and maintain crosscutting concerns. A textual query interface to search for process models or fragments within a repository is proposed in [29], with the aim to assist designers in creating new process models. Finally, in [30] the authors consider a repository of process variants and use reduction techniques to determine the match of variants against a given query. As opposed to our approach, all these approaches do not focus on query efficiency, and do not consider data and resources involved in a process. For this reason, our approach is complementary to them.

To our knowledge, the only work that uses indexing techniques to search for matching process models is [31]. However, here models are represented as annotated finite state automata whereas we use more expressive YAWL models. Moreover, while there is support for a filtering stage, there is no refinement stage, thus reducing the accuracy of the result. The label similarity and the data and resource aspects of a process are also not considered.

8. Conclusion and Future Work

This paper focuses on an efficient method for business process model retrieval. To this end, we borrow the concept of index from the field of graph databases, and use it to speed up query evaluation on large business process model repositories. According to graph database techniques, we follow a two-stage approach for query evaluation. In the first stage (filtering), we obtain an approximate result through the use of indexes. This consists of the set of all process models containing all the tasks in the model query. In the second stage (verification), we refine this set by using an adaptation of Ullman's subgraph isomorphism algorithm, in order to discard those models that do not contain the model query as a subgraph. To make the query results more accurate, we consider the data and resource perspectives together with the control-flow perspective when we check whether a result model contains the model query as a subgraph. To make our approach more generally applicable, we provide support for label similarity. We conducted extensive experiments to demonstrate that the use of these indexes speeds up queries in a significant manner.

An avenue for future work is to apply indexing techniques to improve the efficiency of searching for similar process models. Here the aim is to find all process models in a repository that have a degree of similarity to (but do not necessarily contain) a given model [32].

Acknowledgments

We would like to thank Haiping Zha, Tengfei He, Tao Li and Nianhua Wu for their contribution to the development of BeehiveZ. The work is supported by the National Basic Research Program (973 Plan) of China (No. 2009CB320700), the National High-Tech Development Program (863 Plan) of China (No. 2008AA042301), an NSF Project of China (No. 61003099) and the CSC (China Scholarship Council).

References

[1] D. Shasha, J. T.-L. Wang, R. Giugno, Algorithmics and Applications of Tree and Graph Searching, in: Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002), 2002, pp. 39–52.

[2] X. Yan, P. S. Yu, J. Han, Graph Indexing: A Frequent Structure-based Approach, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2004), 2004, pp. 335–346.

[3] S. Zhang, M. Hu, J. Yang, TreePi: A Novel Graph Indexing Method, in: Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007), 2007, pp. 966–975.

[4] J. Cheng, Y. Ke, W. Ng, A. Lu, Fg-index: towards verification-free query processing on graph databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2007), 2007, pp. 857–872.

[5] D. W. Williams, J. Huan, W. Wang, Graph Database Indexing Using Structured Graph Decomposition, in: Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007), 2007, pp. 976–985.

[6] P. Zhao, J. X. Yu, P. S. Yu, Graph Indexing: Tree + Delta >= Graph, in: Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007), 2007, pp. 938–949.

[7] L. Zou, L. C. 0002, H. Zhang, Y. Lu, Q. Lou, Summarization Graph Indexing: Beyond Frequent Structure-Based Approach, in: 13th International Conference on Database Systems for Advanced Applications (DASFAA 2008), 2008, pp. 141–155.

[8] J. Ullmann, An algorithm for subgraph isomorphism, Journal of the ACM (JACM) 23 (1) (1976) 42.

[9] W. M. P. van der Aalst, A. H. M. ter Hofstede, YAWL: yet another workflow language, Information Systems (IS) 30 (4) (2005) 245–275.

[10] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow Patterns, Distributed and Parallel Databases 14 (1) (2003) 5–51.

[11] G. Decker, R. M. Dijkman, M. Dumas, L. García-Bañuelos, Transforming BPMN Diagrams into YAWL Nets, in: 6th International Conference on Business Process Management (BPM 2008), 2008, pp. 386–389.

[12] A. Brogi, R. Popescu, From BPEL Processes to YAWL Workflows, in: Third International Workshop on Web Services and Formal Methods (WS-FM 2006), 2006, pp. 107–122.

[13] N. Lohmann, E. Verbeek, R. M. Dijkman, Petri Net Transformations for Business Processes - A Survey, Transactions on Petri Nets and Other Models of Concurrency II, Special Issue on Concurrency in Process-Aware Information Systems 2 (2009) 46–63.

[14] T. Jin, J. Wang, N. Wu, M. L. Rosa, A. H. M. ter Hofstede, Efficient and Accurate Retrieval of Business Process Models through Indexing - (Short Paper), in: On the Move to Meaningful Internet Systems: OTM 2010 - Confederated International Conferences: CoopIS, 2010, pp. 402–409.

[15] W. M. P. van der Aalst, The Application of Petri Nets to Workflow Management, Journal of Circuits, Systems, and Computers 8 (1) (1998) 21–66.

[16] A. H. M. Hofstede, W. M. P. Aalst, M. Adams, N. Russell, Modern Business Process Automation: YAWL and its Support Environment, Springer, 2010.

[17] N. Russell, A. H. M. ter Hofstede, newYAWL: Towards Workflow 2.0, Transactions on Petri Nets and Other Models of Concurrency (TOP-NOC) 2 (2009) 79–97.

[18] J. Zobel, A. Moffat, K. Ramamohanarao, Guidelines for Presentation and Comparison of Indexing Techniques, SIGMOD Record 25 (3) (1996) 10–15.

[19] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty Years Of Graph Matching In Pattern Recognition, International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI) 18 (3) (2004) 265–298.

[20] L. Dice, Measures of the amount of ecologic association between species, Ecology 26 (3) (1945) 297–302.

[21] http://lucene.apache.org/.

[22] B. F. van Dongen, A. K. A. de Medeiros, H. M. W. Verbeek, A. J. M. M. Weijters, W. M. P. van der Aalst, The ProM Framework: A New Era in Process Mining Tool Support, in: 26th International Conference on Applications and Theory of Petri Nets (ICATPN 2005), 2005, pp. 444–454.

[23] M. T. Wynn, H. M. W. E. Verbeek, W. M. P. van der Aalst, A. H. M. ter Hofstede, D. Edmond, Reduction rules for YAWL workflows with cancellation regions and OR-joins, Information & Software Technology 51 (6) (2009) 1010–1020.

[24] J. Mendling, H. A. Reijers, W. M. P. van der Aalst, Seven process modeling guidelines (7PMG), Information & Software Technology 52 (2) (2010) 127–136.

[25] H. He, A. K. Singh, Closure-Tree: An Index Structure for Graph Queries, in: Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006), 2006, p. 38.

[26] M. Momotko, K. Subieta, Process Query Language: A Way to Make Workflow Processes More Flexible, in: 8th East European Conference on Advances in Databases and Information Systems (ADBIS 2004), 2004, pp. 306–321.

[27] A. Awad, BPMN-Q: A Language to Query Business Processes, in: Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures (EMISA 2007), 2007, pp. 115–128.

[28] C. D. Francescomarino, P. Tonella, Crosscutting Concern Documentation by Visual Query of Business Processes, in: BPM 2008 International Workshops, 2008, pp. 18–31.

[29] T. Hornung, A. Koschmider, A. Oberweis, A Recommender System for Business Process Models, in: Proceedings of the 17th Workshop on Information Technologies and Systems, 2007.

[30] R. Lu, S. W. Sadiq, Managing Process Variants as an Information Resource, in: 4th International Conference on Business Process Management (BPM 2006), 2006, pp. 426–431.

[31] B. Mahleko, A. Wombacher, Indexing Business Processes based on Annotated Finite State Automata, in: 2006 IEEE International Conference on Web Services (ICWS 2006), 2006, pp. 303–311.

[32] M. Dumas, L. García-Bañuelos, R. M. Dijkman, Similarity Search of Business Process Models, IEEE Data Engineering Bulletin 32 (3) (2009) 23–28.