QUT Digital Repository:
http://eprints.qut.edu.au/

Tao, Jin and Wang, Jianmin and Nianhua, Wu and La Rosa, Marcello and ter Hofstede, Arthur H.M. (2010) ***Efficient and accurate retrieval of business process models through indexing***.

Copyright 2010 The author

Efficient and Accurate Retrieval of Business Process Models through Indexing

Tao Jin[#**], Jianmin Wang[#*], Nianhua Wu[#],
Marcello La Rosa[+], and Arthur H.M. ter Hofstede[+‡**]

[#]School of Software, Tsinghua University,
[*]Department of Computer Science and Technology, Tsinghua University
Beijing, 100084, P. R. China
[+]Queensland University of Technology, Australia
[‡]Eindhoven University of Technology, The Netherlands
{jint05,wnh08}@mails.thu.edu.cn,jimwang@tsinghua.edu.cn,
{m.larosa,a.terhofstede}@qut.edu.au

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 than developing them from scratch. As 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. Experiments are conducted to demonstrate that our proposal achieves a significant reduction in query evaluation time.

Key words: business process model, index, exact query, repository

1 Introduction

Through the application of Business Process Management (BPM), organizations are in a position to rapidly build information systems and to evolve them due to environmental changes, e.g. in legislation or in market demand. Therefore, BPM has matured in recent years and has seen significant uptake in a variety of industries, e.g. health, finance, and manufacturing, and also in government. As a result, in many cases organizations have created large collections of business process models.

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

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

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 process models in the repository that contain this fragment. 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 [1–7], we propose a two-stage approach that reduces the number of models needed to be checked for subgraph isomorphism. First, we filter the model repository through the use of indexes and obtain a set of candidate process models. Second, 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. 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. To demonstrate the effect of the introduction of indexes, and the scalability of our approach, we conduct a series of experiments.

The choice of which process model features to be indexed, and which 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 possible 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 paths as features, as their extraction time is linear on the number of nodes in a model, their storage size is limited, and they can be used to look for isolated nodes (when the path's length is one), and (ii) we store indexes in B+ trees linked to inverted lists, since this structure allows efficient operations and can be updated incrementally.

We assume business processes are modeled as Petri nets or they are mapped from other formalisms into Petri nets, in order to be able to focus on a uniform representation of business processes. In fact, it has been shown that a wide range of business process modeling languages used in practice, e.g. BPMN, EPCs, UML Activity Diagrams and BPEL, or at least significant subsets of them, can be mapped to Petri nets (see [9] for a survey of mappings). In addition, our focus is on the control flow perspective of business process modeling.

The remainder of this paper is organized as follows. Section 2 provides an introduction to Petri nets and formally defines the semantics of a business process model query and the notation of path-based index. Section 3 discusses the construction of indexes while Section 4 shows how these indexes can be used to query a business process model repository. Next, Section 5 analyzes the use of the proposed indexes through a number of experiments. Section 6 discusses related work and Section 7 concludes this paper.

2 Preliminaries

In [10] it was argued that Petri nets offer a number of benefits when used for workflow modeling. Among others, their formal foundation enables verification and the notion of place provides natural support for patterns [11] that require a notion of state, e.g. the deferred choice. Below we formally define the notation of labeled Petri net.

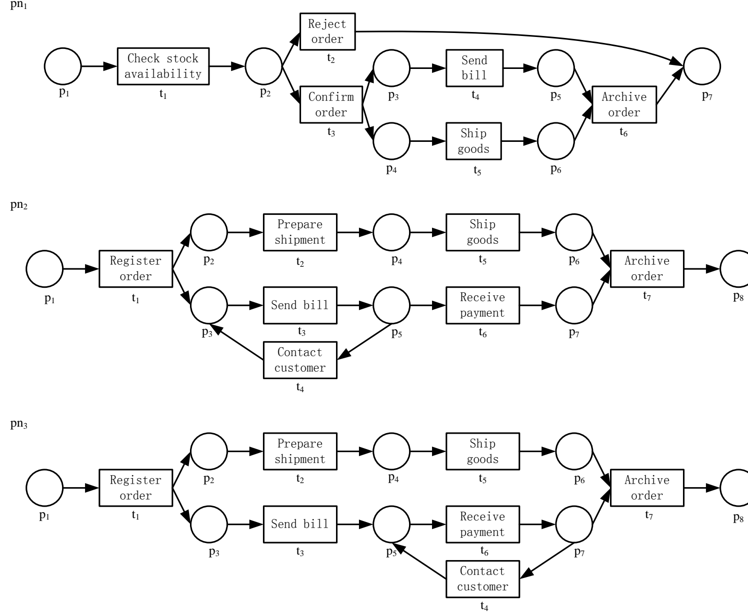

Fig. 1. Business process model examples represented as labeled Petri nets

Definition 1 (Labeled Petri Net). *A labeled Petri net is a tuple (P, T, F, L) where:*

- *P is a finite set of* places*,*
- *T is a finite set of* transitions *$(P \cap T = \varnothing)$,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (the* flow relation*),*
- *$L : T \to V$ is a mapping, where V is a set of labels.*

We use the notation $\bullet t$ to denote the set of input places of transition t, while the notation $t\bullet$ is used to denote the set of output places of transition t. The notations $\bullet p$ and $p\bullet$ have a similar meaning for places. Labels represent the actual activities that are associated with transitions and their introduction helps to formalize the fact that different transitions may perform the same activity. We use $L(t)$ to represent the label of transition t. From now on we use the terms Petri net and labeled Petri net interchangeably.

Example 1. Fig. 1 shows three example business process models represented as Petri nets, where each node is identified by a unique identifier.

A query on a business process model repository is a Petri net, and the result is defined as all Petri nets in the repository that cover that query. First we need to introduce the notion of a net covering another net.

Definition 2 (Petri Net Cover). *Petri net $pn_1 = (P_1, T_1, F_1, L_1)$ is covered by Petri net $pn_2 = (P_2, T_2, F_2, L_2)$, denoted as $pn_1 \sqsubseteq pn_2$, iff there exists a one-to-one function $h : P_1 \to P_2 \cup T_1 \to T_2 \cup F_1 \to F_2$ such that:*

1. *for all $t \in T_1$: $L_1(t) = L_2(h(t))$, i.e. function h preserves transitions' labels;*
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 (Petri Net Query). *Let R be a Petri net model repository and let q be a Petri net query. The result of issuing q over R is $R_q = \{r \in R \mid q \sqsubseteq r\}$.*

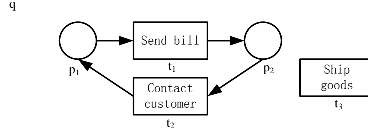

Fig. 2. An example of disconnected Petri net used as a query

Example 2. If we use Petri net q in Fig. 2 as a query and consider the three models of Fig. 1 to constitute the Process model repository R, then only model pn_2 covers q, i.e. $R_q = \{pn_2\}$.

To enhance the efficiency of queries, we use indexes. These indexes are constructed from transition paths.

Definition 4 (Transition Path). *Given a labeled Petri net $pn = (P, T, F, L)$, a sequence of transitions $t_1, \ldots, t_n$ with $n \geq 1$ is a transition path iff $n = 1$ or for all $1 \leq k \leq n-1$ there is place p_k such that $p_k \in t_k \bullet \cap \bullet t_{k+1}$. For $n = 1$ we write the path as $\phi = L(t_1)$, and for $n > 1$ we write the path as $\phi = L(t_1) \to \ldots \to L(t_n)$. The length of this path, $\phi.length$, is n.*

We use LnP_R to denote the set of all transition paths with length n, where $n \geq 1$, in all the models of a repository R. Accordingly, $|LnP_R|$ denotes the size of set LnP_R. If R is clear from the context we will simply write LnP. If m is a process model, the notation LnP_m is equivalent to $LnP_{\{m\}}$. We use $LnCP_R$ to denote the set of all transition paths with lengths up to n, i.e. $LnCP_R = \bigcup_{i=1}^{n} LiP_R$. Therefore, $L1CP_R = L1P_R$, $L2CP_R = L1P_R \cup L2P_R$, etc.

Example 3. In Fig. 2, Petri net q has two paths of length two, i.e. $L2P_q = \{$"*Send bill*" $\to$ "*Contact customer*", "*Contact customer*" $\to$ "*Send bill*"$\}$, and it has three paths of length one, i.e. $L1P_q = \{$"*Send bill*", "*Contact customer*", "*Ship goods*"$\}$.

Definition 5 (Path-based Index). *A path-based index I_R^n of length n in repository R is a set of items of the form $(\phi_i, \phi_i.list)$, where ϕ_i is a transition path of length n in a model of R and $\phi_i.list$ is the set of identifiers of all models that contain ϕ_i.*

The expression *LnP index* denotes a path-based index of length n in repository R. The expression *LnCP index* captures all path-based indexes of lengths upto n in repository R.

3 Index Construction

The choice for particular lengths of path-based indexes may be dependent on the context in which querying takes place. While in our approach indexes of different lengths are stored separately, their structure and the operations upon them are the same. The index storage structure and index operations are the topics of this section.

3.1 LnP Index Structure

To enhance the efficiency of path-based indexes, we do not store the items (ϕ_i,$\phi_i.list$) directly. Instead, we use B+ trees to store items ($\phi_i.hashcode$, $\phi_i.list.pointer$) and use inverted lists to store information about ($\phi_i.list.pointer$, $\phi_i.list$), where $\phi_i.hashcode$ is the hash code of ϕ_i[1] and this code acts as a key in a B+ tree, and $\phi_i.list.pointer$ is a pointer referring to the list $\phi_i.list$. To improve I/O performance, these lists ($\phi_i.list$) are stored as slots of fixed length. When a list requires more than one slot, the first slot contains a reference to the second slot and so on. $\phi_i.list.pointer$ essentially corresponds to a reference to its first slot. It is conceivable that the application of the hash function to two different transition paths yields the same result. Hence leaf entries in these B+ trees correspond to one or more transition paths and provide a reference to a list containing all the models in which these paths occur. This storage structure is illustrated in Fig. 3. In our approach, the roots of B+ trees are kept in memory while the other nodes, as well as the inverted lists, are stored on disk. Cache memory can be used for B+ tree nodes and inverted lists to further improve the efficiency.

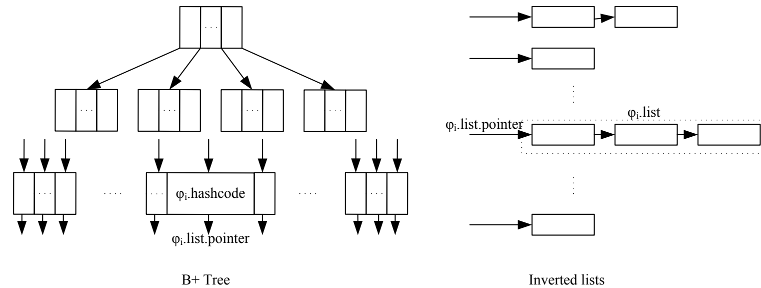

Fig. 3. Index structure used for *LnP index*

As we aim to minimise retrieval time, it is beneficial to keep the depth of B+ trees minimal and to avoid hash collisions as much as possible. Therefore, transition paths of different lengths are indexed in separate B+ trees.

[1] For a path of length more than one the hascode is computed on the string "$L(t_1)->\ldots->L(t_n)$".

Example 4. Given the model repository shown in Fig. 1, we can obtain all paths of length one and calculate their hash codes. Some of these paths and the corresponding hash codes are shown in Table 1. Then an *L1P index* can be created based on this data.

Table 1. A sample of indexed items for paths of length one in Fig. 1

ϕ_i	$\phi_i.hashcode$	$\phi_i.list.pointer$	$\phi_i.list$
"Send bill"	404840567069183	4	pn_1, pn_2, pn_3
"Ship goods"	9582605794425063	5	pn_1, pn_2, pn_3
"Contact customer"	-8147188639381854257	9	pn_2, pn_3
⋮	⋮	⋮	⋮

3.2 LnP Index Creation

We now present a generic algorithm for the creation of *LnP* indexes. This algorithm is described in pseudocode as Algorithm 1. Line 1 extracts all the transition paths of length n from the given model pn. In lines 2-6 all transition paths are processed sequentially. Line 3 computes the $\phi_i.hashcode$ of path ϕ_i. Line 4 adds the $\phi_i.hashcode$ to the B+ tree and obtains the $\phi_i.list.pointer$ to the inverted list. Line 5 adds the identifier of model pn to the inverted list $\phi_i.list$ found at the address provided by $\phi_i.list.pointer$. The details of the functions called in this algorithm are explained subsequently.

Algorithm 1: PathIndexConstruction

```
    input: pn: a Petri net
           n: the length of the path
1.1 sps ← PathExtraction(pn,n);
1.2 foreach path in sps do
1.3 |   hashcode ← DEKHash(path);
1.4 |   sid ← AddToBplusTree(hashcode);
1.5 |   AddToInvertedList(sid, pn);
1.6 end
```

Path Extraction According to Definition 4, *LnP* for $n \geq 1$ can be derived from a process model in a straightforward manner. This is achieved by application of the Function `PathExtraction`.

Hash the Path String As labels of transition paths may be arbitrarily long strings, a hash function is applied to map such strings to numbers in order to save space and speed up querying. The hash function `DEKHash` used in this paper is that proposed by D.E. Knuth in [12].

Map Hash Code to Inverted List Function `AddToBplusTree` shown in Algorithm 2 yields the position ($\phi_i.list.pointer$) of the first slot of the list $\phi_i.list$ of process models in which a given transition path ϕ_i, specified by its hashcode $\phi_i.hashcode$, occurs. If $\phi_i.hashcode$ already exists in the B+ tree, $\phi_i.list.pointer$

is returned immediately, as described in Lines 2-4. Otherwise the first slot for a new inverted list is created and the pointer to this slot is returned, as described in Lines 5-9. Here, Functions `BplusTreeSearch` and `BplusTreeAdd` are generic operations on B+ trees, used for querying and adding nodes to such trees respectively. Function `AllocateNewSlot` is used to allocate disk storage for an inverted list and returns the position of an available slot for this list.

Algorithm 2: AddToBplusTree(hashcode)

```
     input  : hashcode: the hashcode of a path
     output: sid: the inverted list id
2.1  sid ← BplusTreeSearch(hashcode);
2.2  if sid is not null then
2.3  |  return sid;
2.4  end
2.5  else
2.6  |  sid ← AllocateNewSlot();
2.7  |  BplusTreeAdd(hashcode, sid);
2.8  |  return sid;
2.9  end
```

Update the Inverted List In the next step, as described in Algorithm 3, the identifier of a process model is added to the list $\phi_i.list$ where the transition path ϕ_i that is currently considered occurs. This may lead to a slot overflow in which case a new slot needs to be created as well as a reference from the current slot to this new slot.

Algorithm 3: AddToInvertedList(sid, pn)

```
     input: sid: the inverted list id
            pn: the process model
3.1  pns ← get the sid inverted list;
3.2  if pns not contains pn.id then
3.3  |  add pn.id into pns;
3.4  end
```

3.3 LnP Index Updating

As can be seen in Algorithm 1, when a new model is added to the repository, the *LnP index* can be updated incrementally (when multiple indexes are used, all these indexes will be updated). When models are deleted from the repository, which rarely happens in a retrieval system, we can use a list to record the identifiers of these models rather than remove them immediately from the index structure as this would not be very efficient. When the number of models recorded in this list exceeds a certain threshold, we can construct the index from scratch.

4 Query Processing

First, we give an overview of query processing in Section 4.1. Then, we show how a query can be processed in two stages. These stages are discussed in Section 4.2 and Section 4.3.

4.1 Overview of Query Processing

Petri net query processing is divided into two stages. The first stage of query processing only acts as a filter. The first reason is that the result from the first stage may not be exact due to hash collisions where different paths may be mapped to the same list of process models. The second reason is that by focussing on transition paths context information is not taken into account, e.g. the fact that a choice may exist between two subsequent transitions in a model is lost as well as the order of such paths 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.

When multiple indexes are available during the first stage of query processing, those indexes whose lengths correspond to paths in the query can be exploited to obtain a set of candidate models. It is best to try to use an index with the largest length first for those transition paths that match the length of this index. During this process we can determine those transitions that do not occur in any path of this length. These transitions can then be used when checking another index with the next maximal length and so on.

Example 5. Consider again Petri net q of Fig. 2. Assuming that we have constructed two indexes: an *L1P index* and an *L2P index*, i.e. that we have constructed an *L2CP index*, first the *L2P index* is applied and then the *L1P index* is applied as transition t_3 is not used in any transition path of length two in the query. If however transition t_3 is removed from the query, the *L1P index* need not be applied at all.

The procedure for query processing is described in Algorithm 4. Line 1 extracts all transition paths of some length from the Petri net acting as the query. For example, if the *L1P index* is used, all paths of length one are extracted. If the *L2CP index* is used, all paths of length two are extracted, and then all paths of length one for those transitions that have not been used in any transition path of length two are also extracted. Lines 2-6 work as a filter and a set of candidate models is obtained, that is, the first stage of query processing. Given a path ϕ_i, Line 3 obtains a list of models $\phi_i.list$ containing this path using the index whose length equals $\phi_i.length$. In Line 6 the intersection of the resulting candidate sets is computed so as to retain only those candidate models that contain all the path queries. If some models were deleted from the repository, we would need to remove them from this intersection (this is not shown). In Lines 7-11 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 8, the function `CBMTest`, explained later, is applied to eliminate the effect of hash collisions and take the context of these paths into account.

Algorithm 4: PetriNetQuery

```
     input  : pn: the Petri net query
     output: R: a set of Petri nets cover pn
4.1  sps ← PathExtraction(pn);
4.2  foreach path ∈ sps do
4.3  |  list ← PathIndexQuery(path);
4.4  |  add list to lists;
4.5  end
4.6  R ← Intersection(lists);
4.7  foreach respn ∈ R do
4.8  |  if CBMTest(pn, respn) fails then
4.9  |  |  delete respn from R;
4.10 |  end
4.11 end
4.12 return R;
```

4.2 Use Path-based Indexes as Filter

Function `PathIndexQuery` shown in Algorithm 5 is used to retrieve all models in which a certain transition path occurs. Given ϕ_i, Line 1 computes its hash code $\phi_i.hashcode$. Line 2 searches in the applicable B+ tree, i.e. the one that stores transition paths of the same length as ϕ_i, using this hash code to get $\phi_i.list.pointer$, Then Line 3 retrieves the list of models $\phi_i.list$.

Algorithm 5: PathIndexQuery(path)

```
     input  : path: a path
     output: list: a set of Petri nets containing path
5.1  hashcode ← DEKHash(path);
5.2  sid ← BplusTreeSearch(hashcode);
5.3  list ← get the sid inverted list from the inverted lists;
5.4  return list;
```

4.3 Refine the Result

Function `CBMTest` implements the Petri net cover check according to Definition 2. It is an adaptation of Ullman's graph isomorphism algorithm [8] to Petri nets. In [13] 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 Petri nets.

In `CBMTest`, transition t can be mapped to transition t' if and only if t and t' share the same label. Place p can be mapped to place p' if and only if there is a one-to-one mapping between the input transitions of p and p' and between the output transitions of p and p'. `CBMTest` tries to find a one-to-one mapping between the nodes of the Petri net query and the candidate Petri net being

investigated. All arcs in the Petri net query must be preserved in the candidate Petri net. If this is impossible, the candidate Petri net is removed from the result set.

Example 6. Consider q in Fig. 2 as a Petri net query. Consider pn_3 in Fig. 1 as a candidate Petri net. Transitions t_1, t_2 and t_3 in q can be mapped to t_3, t_4 and t_5 in pn_3 respectively. But p_1 in q cannot be mapped to p_3 or p_5 in pn_3. Hence q cannot be covered by pn_3. On the other hand, consider pn_2 in Fig. 1 as a candidate Petri net. In that case, t_1, t_2 and t_3 in q can be mapped to t_3, t_4 and t_5 in pn_2 respectively, while p_1 and p_2 in q can be mapped to p_3 and p_5 in pn_2 respectively. At the same time, the arcs $< p_1, t_1 >$, $< t_1, p_2 >$, $< p_2, t_2 >$, and $< t_2, p_1 >$ in q are all preserved in pn_2. So pn_2 covers q.

Example 7. Consider the model repository $R = \{pn_1, pn_2, pn_3\}$ as shown in Fig. 1. We use Petri net q in Fig. 2 as a Petri net query and assume that the *L1P index* is used. First, we extract all transition paths of length one from q. This yields three paths, ϕ_1^q = "*Send bill*", ϕ_2^q = "*Contact customer*" and ϕ_3^q = "*Ship goods*". Using the DEK hash function, we can get $\phi_1^q.hashcode = 404840567069183$, $\phi_2^q.hashcode = -8147188639381854257$ and $\phi_3^q.hashcode = 9582605794425063$. Then we can query the repository using this index to obtain $\phi_1^q.list = \{pn_1, pn_2, pn_3\}$, $\phi_2^q.list = \{pn_2, pn_3\}$ and $\phi_3^q.list=\{pn_1,pn_2,pn_3\}$. These sets correspond to those shown in Table 1. After taking the intersection of these sets, we get the candidate model set $R_q^c = \{pn_2, pn_3\}$. 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 $R_q = \{pn_2\}$.

5 Tool Support and Evaluation

In order to validate our approach, We developed a system called BeehiveZ[2]. BeehiveZ is a java application, which makes use of the Derby RDBMS to store process models as data type CLOB. Based on the generic *LnP index*, we implemented the *L1P index* and *L2CP index*. The ProM [14] library was used for the representation and display of Petri nets.

We conducted a number of experiments to determine the effectiveness and efficiency of the algorithms presented in the previous section by using a computer with Intel(R) Core(TM)2 Duo CPU E8400 @3.00GHz and 3.21GB memory. This computer ran Windows XP Professional SP3 and jdk6. All models were generated automatically using an algorithm that produces a collection of Petri nets randomly. The rules used in our generator come from [15].

5.1 Effectiveness

In order to validate the effectiveness of our approach, every time a model was added to the repository, we also used the model as a query on the repository. We

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

checked whether the resulting set of process models indeed contained that model. We also carried out a second automated test to determine the effectiveness of our approach. This consisted of the generation of a set of Petri nets Q, the subsequent generation of another set of Petri nets Δ, typically much larger, and the execution as a query of every Petri net $q \in Q$ over the collection of Petri nets $Q \cup \Delta$, to check whether the resulting sets always contained q.

Through these test approaches, experiments show that the *L1P index* and the *L2CP index* never exclude models that satisfy the query. If only the *L2P index* is used, this result does not hold as the query may contain isolated transitions. In order to satisfy the fourth requirement proposed in the introduction, i.e. any process model fragment should be allowed as a query, *L1P* must be indexed.

5.2 Efficiency

To evaluate the efficiency of our approach, we conducted further experiments. In these experiments 10 models were generated to act as queries and more than 40,000 models were generated to populate the business process model repository. The 10 queries were evaluated each time after the addition of a certain number of freshly generated process models. Table 2 shows the characteristics of the various queries q_i, $1 \leq i \leq 10$. Specifically, for repository R, there were 40,210 Petri nets, the number of transitions in the various models ranged from 1 to 198, the number of places from 2 to 140, the number of arcs from 2 to 765, and there were at most 242,234 differently labeled transitions out of 2,201,573 transitions in total.

Table 2. Characteristics of model queries

	Number of transitions	Number of places	Number of arcs
q_1	1	2	2
q_2	21	15	48
q_3	41	26	107
q_4	61	36	223
q_5	81	66	215
q_6	101	72	271
q_7	121	80	311
q_8	141	89	443
q_9	161	96	469
q_{10}	181	106	491

We define R_q^c as the candidate set of answers resulting from q applied to the repository R using an 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 Petri net 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 1.

$$T_q = T_s + |R_q^c| \times T_{I/O} + |R_q^c| \times T_v. \quad (1)$$

Equation 2 provides the query response time when no index is used.

$$T_q = |R| \times T_{I/O} + |R| \times T_v. \quad (2)$$

As $|R_q^c|$ is often much smaller than $|R|$, the use of an index can save a significant amount of time, as shown in Fig. 4(a). The size of $|R_q^c|$ and $|R|$ recorded in our experiments can be found in Table 3. We found that the candidate set size for q_i, $2 \leq i \leq 10$, is always one whichever path-based index is used, which means that $|R_q^c| = |R_q^f|$ during the processing of these queries. Hence, it can be observed that the filtering power of the path-based indexes for these queries is good.

Table 3. The size of candidate set

	Size of model repository	Size of candidate set using *L1P index* or *L2CP index*
q_1	210	1
q_1	10210	15
q_1	20210	32
q_1	30210	38
q_1	40210	45
q_{2-10}	$210 - 40210$	1

Fig. 4(c) shows the search time for queries q_2 and q_4 using both the *L1P index* and the *L2CP index*. It can be observed that the performance of one index is not always better than the other one.

For the *LnP index*, T_s can be calculated according to Equation 3.

$$T_s = T_e \times |\Phi_q| + f(|\Phi_R|) \times |\Phi_q|. \quad (3)$$

Here, T_e denotes the extraction time of one path with length n from a model. $|\Phi_q|$ denotes the number of paths with length n in Petri net query q. This number determines how many paths will be extracted from q and how many times the B+ tree will be traversed. $|\Phi_R|$ is the number of paths with length n in repository R. This number determines the depth of the B+ tree. $f(|\Phi_R|)$ denotes the time it takes for a single B+ tree traversal. $|\Phi_q|$ and $|\Phi_R|$ are determined by the characteristics of query q and repository R, which explains why in some cases a particular index for a particular query on a certain repository performs better than another index for that query on that repository, while for another query the converse applies. When n is larger, T_e is larger, but $|\Phi_q|$ and $|\Phi_R|$ may be smaller.

Fig. 4(d) shows the time required for index construction. Here it can be observed that the *L1P index* performs better than the *L2CP index*. Whenever a new model is added to the repository, the index does not need to be reconstructed

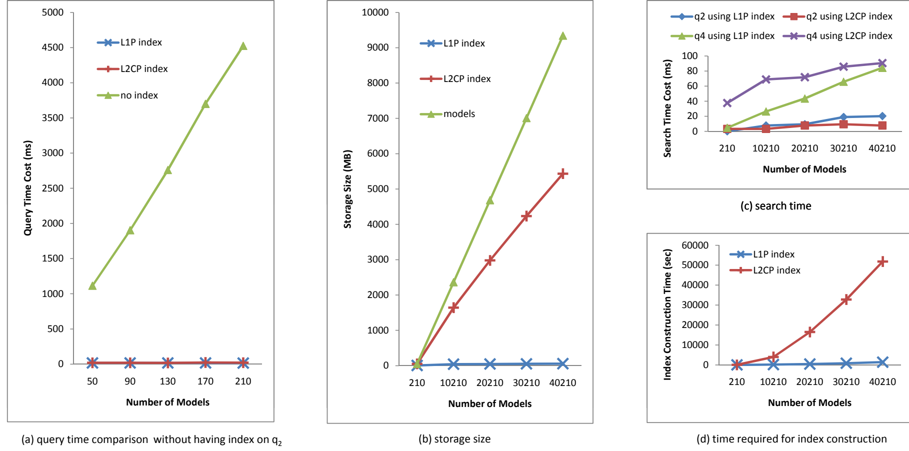

Fig. 4. Performance of path-based indexes

but can simply be modified . Therefore the time required for index construction is acceptable. The storage size of path-based indexes is shown in Fig. 4(b). Here too the *L1P index* performs better than the *L2CP index*.

From Fig. 4, one can deduce that for the *LnCP index* if the length of paths is too large, the storage size may exceed the size required for storing the models, and the index construction would be more time consuming. This is a consequence of the fact that there are more paths in the model. Therefore the *L1P index* may be more suitable in practice.

In the above experiments, the order of B+ trees was 100, every inverted list slot contained at most 100 model identifiers, and caching for B+ trees and inverted lists was disabled. The performance of path-based indexes would be enhanced if we used cache for B+ trees and inverted lists. The size of cache for B+ trees and inverted lists can be configured in the BeehiveZ system, and the *LRU* (Least Recently Used) algorithm is used. For example, by caching 500 B+ tree nodes and 1,000 inverted lists in memory for the *L2CP index*, the performance of q_i for all $1 \leq i \leq 10$ on model repository R ($|R| = 40,210$) improves on average by 16%. Naturally, caching will not necessarily always improve performances.

5.3 Discussion

It is difficult to answer the question which path-based index, or combination of path-based indexes, performs the best as this depends on the characteristics of the models in the repository, the queries, and also the user's objective. We must find a tradeoff among the index construction time, index storage size, path extraction time and query processing time using the index. Our own preference is to use the *L1P index*, because it is simple, and from the above experiments we can see that it satisfies the requirements proposed in the introduction well. It is possible to use more than one index and again, depending on the circumstances, this may offer benefits. For example, if in the majority of Petri net queries all transitions occur in transition paths with length of at least three, but there may

be queries with isolated transitions, it may be preferable to use the $L3P$ index in combination with the $L1P$ index.

6 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 (similar to our transition 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. All these approaches work on abstract graphs with one type of node, while our approach operates on Petri nets which are bipartite graphs. In [2–4, 6], focus is on frequent sub-structures and thus they cannot efficiently deal with queries consisting of isolated nodes or infrequent sub-structures. Thus they are not suitable to work with process models, 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. Therefore, according to the requirements proposed in the introduction, we believe it is better to use paths as an indexing feature for process models.

Our work has also commonalities with query languages for process models. For example, the Business Process Query Language (BPQL) [16] 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 [17] 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 [18], 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 [19], with the aim to assist designers in creating new process models. Finally, in [20] 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. For this reason, our approach is complementary to the above ones.

7 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. We propose path-based indexes to speed up queries on 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 transition paths in the model query. In the second stage (verification), we refine this set by using an adaptation of Ullman's subgraph isomorphism algorithm to Petri nets, in order to discard those models that do not contain the model query as a subgraph. We conducted extensive experiments to demonstrate that the use of these indexes speeds up queries in a significant manner.

The current approach has some limitations which provide scope for future work. First, we assume process models to be represented as Petri nets. The approach could be further improved by eliminating this dependency and employing a *canonical* process format such as that defined in the APROMORE project [21]. Second, focus is solely on the control flow perspective of business processes. It would be interesting to enrich the queries with information concerning data manipulation and resource allocation.

Another 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 most closely resemble (but not necessarily contain) a given process model which is used as query [22].

Acknowledgments. We thank Lijie Wen for his helpful suggestions to this paper. We also thank Haiping Zha, Tengfei He and Tao Li for their contribution to the development of BeehiveZ. This paper is partially supported by the 973 Project of China (No.20081970189) and CSC (China Scholarship Council).

References

1. D. Shasha, J.T-L. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In *PODS*, pages 39–52, 2002.
2. X. Yan, Ph.S. Yu, and J. Han. Graph Indexing: A Frequent Structure-based Approach. In *SIGMOD*, pages 335–346, 2004.
3. S. Zhang, M. Hu, and J. Yang. TreePi: A Novel Graph Indexing Method. In *ICDE*, pages 966–975, 2007.
4. J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: Towards Verification-free Query Processing on Graph Databases. In *SIGMOD*, pages 857–872, 2007.

5. D.W. Williams, J. Huan, and W. Wang. Graph Database Indexing Using Structured Graph Decomposition. In *ICDE*, pages 976–985, 2007.
6. P. Zhao, J.X. Yu, and Ph.S. Yu. Graph Indexing: Tree + Delta >= Graph. In *VLDB*, pages 938–949, 2007.
7. L. Zou, L. Chen, H. Zhang, Y. Lu, and Q. Lou. Summarization Graph Indexing: Beyond Frequent Structure-Based Approach. In *DASFAA*, pages 141–155, 2008.
8. J.R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.
9. N. Lohmann, E. Verbeek, and R.M. Dijkman. Petri Net Transformations for Business Processes - A Survey. *Transactions on Petri Nets and Other Models of Concurrency*, 2:46–63, 2009.
10. W.M.P. Van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits Systems and Computers*, 8(1):21–66, FEB 1998.
11. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
12. D.E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Professional, 1998.
13. D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty Years Of Graph Matching In Pattern Recognition. *IJPRAI*, 18(3):265–298, 2004.
14. B.F. van Dongen, A.K.A. de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM Framework: A New Era in Process Mining Tool Support. In *ICATPN*, pages 444–454, 2005.
15. P. Chrzastowski-Wachtel, B. Benatallah, R. Hamadi, M. O'Dell, and A. Susanto. A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling. In *Business Process Management*, pages 336–353, 2003.
16. M. Momotko and K. Subieta. Process Query Language: A Way to Make Workflow Processes More Flexible. In *ADBIS*, pages 306–321, 2004.
17. A. Awad. BPMN-Q: A Language to Query Business Processes. In *EMISA*, pages 115–128, 2007.
18. C. Di Francescomarino and P. Tonella. Crosscutting Concern Documentation by Visual Query of Business Processes. In *Business Process Management Workshops*, pages 18–31, 2008.
19. T. Hornung, A. Koschmider, and A. Oberweis. A Recommender System for Business Process Models. In *Proceedings of the 17th Workshop on Information Technologies and Systems*, 2007.
20. R. Lu and S.W. Sadiq. Managing Process Variants as an Information Resource. In *Business Process Management*, pages 426–431, 2006.
21. M. La Rosa, H.A. Reijers, W.M.P. van der Aalst, R.M. Dijkman, J. Mendling, M. Dumas, and L. Garcia-Banuelos. APROMORE : An Advanced Process Model Repository. QUT ePrints, http://eprints.qut.edu.au/27448/, 2009.
22. M. Dumas, L. Garcia-Banuelos, and R. Dijkman. Similarity Search of Business Process Models. *Bulletin of the Technical Committee on Data Engineering*, 32(3):25–30, September 2009.