Efficient and Accurate Retrieval of Business Process Models through Indexing
(Short Paper)

Tao Jin[1,2,⋆], Jianmin Wang[1,2], Nianhua Wu[2], Marcello La Rosa[3], and Arthur H.M. ter Hofstede[3,4,⋆⋆]

[1] Department of Computer Science and Technology, Tsinghua University, China
[2] School of Software, Tsinghua University, China
[3] Queensland University of Technology, Australia
[4] 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 various 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. 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.

Keywords: 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. Therefore, BPM has matured in recent years and has seen significant uptake in a variety of industries and also in government. This has led to the proliferation of large collections of business process models.

Managing such 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 providing efficient support for querying business process model repositories. Given a process fragment (the model query), we are

⋆ Visit to QUT funded by China Scholarship Council.
⋆⋆ Senior Visiting Scholar of Tsinghua University.

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 [1,2,3,4,5,6,7], we propose a two-stage approach that reduces the number of models needed to be checked for subgraph isomorphism. 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] to refine the set of candidate models by extracting 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.

In order to be able to focus on a uniform representation of business processes, we assume business processes are modeled as Petri nets or mapped from other formalisms into Petri nets. 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 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 our approach through experiments. Section 6 discusses related work and Section 7 concludes this paper. For a more detailed treatment of our work, we refer the reader to [10].

2 Preliminaries

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.

Definition 1 (Petri Net Cover). *Labeled Petri net $pn_1 = (P_1, T_1, F_1, L_1)$ is covered by labeled 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 transition 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.*

Let R be a Petri net model repository and let q be a Petri net query. The result of issuing q over R is $R_q = \{r \in R \mid q \sqsubseteq r\}$. To enhance the efficiency of queries, we use indexes. These indexes are constructed from transition paths.

Definition 2 (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 $\phi = L(t_1) \rightarrow \ldots \rightarrow L(t_n)$. The length of this path, ϕ.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. 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. $LnCP_R = \bigcup_{i=1}^{n} LiP_R$.

Definition 3 (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

To enhance the efficiency of path-based indexes, we do not store the items (ϕ_i,ϕ_i.*list*) directly. Instead, we use B+ trees to store items (ϕ_i.*hashcode*, ϕ_i.*list.pointer*) and use inverted lists to store information about (ϕ_i.*list.pointer*, ϕ_i.*list*), where ϕ_i.*hashcode* is the hash code of ϕ_i[1] and this code acts as a key in a B+ tree, and ϕ_i.*list.pointer* is a pointer referring to the list ϕ_i.*list*. To improve I/O performance, these lists (ϕ_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. ϕ_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. 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 in different files. Cache memory can be used for B+ tree nodes and inverted lists to further improve the efficiency.

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 (since any process model fragment should be allowed as a query, *L1P* must be indexed).

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 ϕ_i.*hashcode* of path ϕ_i. Line 4 adds the ϕ_i.*hashcode* to the B+ tree and obtains the ϕ_i.*list.pointer* to

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

Algorithm 1. PathIndexConstruction

```
input: pn: a Petri net
       n: the length of the path
1 sps ← PathExtraction(pn,n);
2 foreach path in sps do
3     hashcode ← DEKHash(path);
4     sid ← AddToBplusTree(hashcode);
5     AddToInvertedList(sid, pn);
6 end
```

the inverted list. Line 5 adds the identifier of model *pn* to the inverted list $\phi_i.list$ found at the address provided by $\phi_i.list.pointer$.

As shown 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 a model is updated, the old version is deleted and the new version is added. 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 reconstruct the index from scratch.

4 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.

The procedure for query processing is described in Algorithm 2. 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

Algorithm 2. PetriNetQuery

```
input  : pn: the Petri net query
output: R: a set of Petri nets cover pn
1  sps ← PathExtraction(pn);
2  foreach path ∈ sps do
3      list ← PathIndexQuery(path);
4      add list to lists;
5  end
6  R ← Intersection(lists);
7  foreach respn ∈ R do
8      if CBMTest(pn, respn) fails then
9          delete respn from R;
10     end
11 end
12 return R;
```

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. Function `CBMTest` implements the Petri net cover check according to Definition 1. It is an adaptation of Ullman's graph isomorphism algorithm [8] to Petri nets.

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 library was used for the representation and display of Petri nets.

We conducted a number of experiments, both on a synthetic dataset and on a dataset consisting of SAP Reference Models (for this latter experiment we refer to [10]) to determine efficiency of the algorithms presented in the previous section. 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. All synthetic models were generated automatically using an algorithm

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

that produces a collection of Petri nets randomly. The rules used in our generator come from [11]. In the experiments on the synthetic dataset 10 models were generated to act as queries and more than 40,000 models were generated to populate the business process model repository. 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. The 10 queries were evaluated each time after the addition of a certain number of freshly generated process models.

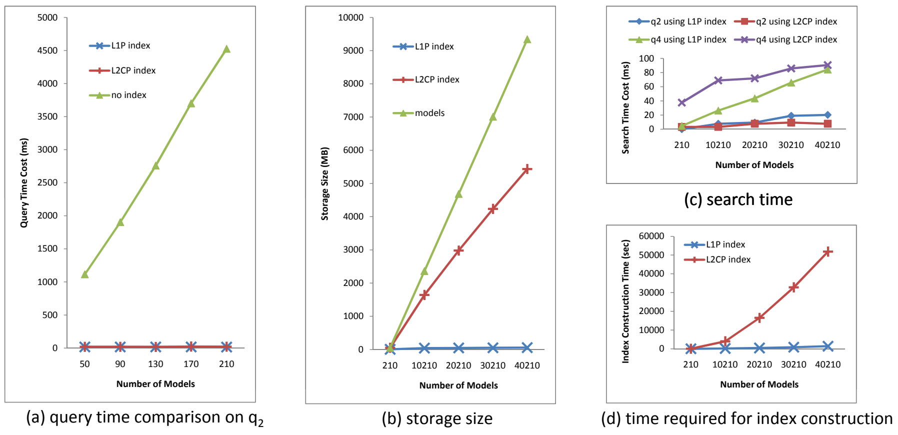

Fig. 1. Performance of path-based indexes

As the number of candidate models is often much smaller than the size of the repository, the use of an index can save a significant amount of time, as shown in Fig. 1(a). Fig. 1(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. Fig. 1(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 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. 1(b). Here too the *L1P index* performs better than the *L2CP index*. From Fig. 1, 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 models. 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.

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 [1,2,3,4,5,6,7] use different graph features as indexes. All these approaches work on abstract graphs with one type of node, while our approach operates on Petri nets which are bipartite graphs. 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. Hence, they are not suitable for process models, where each task may have a different label (thus reducing the frequency of common sub-structures) 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 [12,13,14,15]. As opposed to our approach, all these approaches do not focus on query efficiency. For this reason, our approach is complementary to these approaches.

An indexing technique is used in [16] to search for matching process models. However, here models are represented as annotated finite state automata whereas we use generic Petri nets. Moreover, while there is support for a filtering stage, there is no refinement stage, thus reducing the accuracy of the result.

7 Conclusion and Future Work

We propose path-based indexes to speed up queries on business process model repositories in this paper. We follow a two-stage approach for query evaluation. In the first stage (filtering), we obtain an approximate result through the use of indexes. 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 experiments to demonstrate that the use of these indexes speeds up queries in a significant manner.

In this paper, our focus is solely on the control flow perspective of business processes. It would be interesting to enrich queries with information concerning data manipulation and resource allocation in the future.

Acknowledgments. We thank Lijie Wen for his helpful suggestions. We also thank Haiping Zha, Tengfei He and Tao Li for their contribution to the development of BeehiveZ. The work is supported by the National Basic Research Program (973 Plan) of China (No. 2009CB320700 and No. 2007CB310802), the National High-Tech Development Program (863 Plan) of China (No. 2008AA042301 and No. 2007AA040607) and CSC (China Scholarship Council).

References

1. Shasha, D., Wang, J.T.-L., Giugno, R.: Algorithmics and Applications of Tree and Graph Searching. In: PODS, pp. 39–52 (2002)
2. Yan, X., Yu, P.S., Han, J.: Graph Indexing: A Frequent Structure-based Approach. In: SIGMOD, pp. 335–346 (2004)
3. Zhang, S., Hu, M., Yang, J.: TreePi: A Novel Graph Indexing Method. In: ICDE, pp. 966–975 (2007)
4. Cheng, J., Ke, Y., Ng, W., Lu, A.: Fg-index: Towards Verification-free Query Processing on Graph Databases. In: SIGMOD, pp. 857–872 (2007)
5. Williams, D.W., Huan, J., Wang, W.: Graph Database Indexing Using Structured Graph Decomposition. In: ICDE, pp. 976–985 (2007)
6. Zhao, P., Yu, J.X., Yu, P.S.: Graph Indexing: Tree + Delta >= Graph. In: VLDB, pp. 938–949 (2007)
7. Zou, L., Chen, L., Zhang, H., Lu, Y., Lou, Q.: Summarization Graph Indexing: Beyond Frequent Structure-Based Approach. In: Haritsa, J.R., Kotagiri, R., Pudi, V. (eds.) DASFAA 2008. LNCS, vol. 4947, pp. 141–155. Springer, Heidelberg (2008)
8. Ullmann, J.R.: An Algorithm for Subgraph Isomorphism. Journal of the ACM 23(1), 31–42 (1976)
9. Lohmann, N., Verbeek, E., Dijkman, R.M.: Petri Net Transformations for Business Processes - A Survey. Transactions on Petri Nets and Other Models of Concurrency 2, 46–63 (2009)
10. Jin, T., Wang, J., Wu, N., La Rosa, M., ter Hofstede, A.H.M.: Efficient and accurate retrieval of business process models through indexing. Technical report, Tsinghua University (2010)
11. Chrzastowski-Wachtel, P., Benatallah, B., Hamadi, R., O'Dell, M., Susanto, A.: A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 336–353. Springer, Heidelberg (2003)
12. Momotko, M., Subieta, K.: Process Query Language: A Way to Make Workflow Processes More Flexible. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 306–321. Springer, Heidelberg (2004)
13. Awad, A.: BPMN-Q: A Language to Query Business Processes. In: EMISA, pp. 115–128 (2007)
14. Di Francescomarino, C., Tonella, P.: Crosscutting Concern Documentation by Visual Query of Business Processes. In: Business Process Management Workshops, pp. 18–31 (2008)
15. Hornung, T., Koschmider, A., Oberweis, A.: A Recommender System for Business Process Models. In: Proceedings of the 17th Workshop on Information Technologies and Systems (2007)
16. Mahleko, B., Wombacher, A.: Indexing business processes based on annotated finite state automata. In: ICWS, pp. 303–311 (2006)