TOP-K Projection Queries for Probabilistic Business Processes

Daniel Deutch Tova Milo
{danielde,milo}@cs.tau.ac.il

ABSTRACT

A Business Process (BP) consists of some business activities undertaken by one or more organizations in pursuit of some business goal. Tools for querying and analyzing BP specifications are extremely valuable for companies. In particular, given a BP specification, identifying the top-k flows that are most likely to occur in practice, out of those satisfying a given query criteria, is crucial for various applications such as personalized advertizement and BP web-site design.

This paper studies, for the first time, top-k query evaluation for queries with *projection* in this context. We analyze the complexity of the problem for different classes of distribution functions for the flows likelihood, and provide efficient (PTIME) algorithms whenever possible. Furthermore, we show an interesting application of our algorithms to the analysis of BP execution traces (logs), for recovering missing information about the run-time process behavior, that has not been recorded in the logs.

1. INTRODUCTION

A Business Process (BP for short) is a collection of logically related activities that, when combined in a flow, achieve a business goal. To simplify software development, it is a common practice to provide a high level description of the BP operational flow (using a standard specification language such as BPEL [5]), and then have the software be automatically generated from this specification. Since the BP logic is captured by the specification, tools for querying and analyzing BP specifications are extremely valuable for companies [2]. They allow to optimize the BP, identify potential problems, and reduce operational costs.

A typical query engine is given as input a BP specification and an execution pattern of interest, and identifies, among the potential execution flows of the BP, all those (sub)flows having the structure specified by the pattern ([2, 3, 13]). Note, however, that among all possible flows, some are typically more interesting than others. In particular, given a BP specification, identifying the top-k query answers (flows) that are most likely to occur in practice, is crucial for various applications. It can be used, for instance, to adjust the BP web-site design to the needs of certain user groups, or to personalize on-line advertisements. The importance of top-k query evaluation is enhanced by the fact that the number of answers (qualifying sub-flows) to a simple query may be extensively large, or even infinite when the BP contains recursion [2].

As a simple example, consider a BP of an on-line, Web-based travel agency. An analyzer that wishes to examine the BP execution flows may issue queries such as “In what ways may one reserve a travel package containing a flight and an hotel?”, or “How is this done for travellers of a particular airline company, say British Airways?”. There may be many ways to book such travel packages. But assume, for instance, that we obtain that a popular scenario for British Airways reservations is one where users first search for a package containing both flights and hotels, but eventually book a British Airways flight without reserving an hotel. Such a result may imply that the combined deals suggested for British Airways fliers are unappealing, (as users are specifically interested in such deals, but refuse those presented to them), and can be used to improve the Web site.

In a previous paper [14] we have suggested a model for ranking of BP execution flows based on their likelihood of occurrence, and presented an efficient top-k query evaluation algorithm for *selection queries*. Namely, queries that given an execution pattern and a BP specification, retrieve the top-k *full* (start-to-end) possible execution flows which contain some occurrence of the pattern. In many cases, however, the user may not be interested in the whole flow (which may be long and complex) but only in *specific parts* which are relevant to her analysis (e.g. only the sub-flow dealing with hotel reservation, or only the activities issued when a payment request is rejected). To address this need we consider in this paper *projection queries*, which retrieve the sub-flows of interest, out of the full qualifying flows, and provide an efficient algorithm for top-k evaluation of such queries.

An important question that rises when considering projection queries is the choice of a ranking metric for the query results. Utilizing the notion of flow likelihood, we define *rank* for projection results, as the *maximal* likelihood of a flow containing the projection as a sub-flow. (Other possible rankings are considered in future work). We will see

ICDT 2009, March 23–25, 2009, Saint Petersburg, Russia.

that although there is a tight relationship between the semantics of selection and projection queries (both searching for flows with maximal likelihoods), new techniques need to be developed to provide efficient (PTIME) query evaluation for projection queries. Indeed, we show that the standard, common, use of selection as an intermediate input for projection yields in this case an exponential blow-up. Instead, our novel query evaluation algorithm constructs a direct, *compact representation* of the top-k projections, avoiding materialization of the intermediate (full) qualifying flows. *This algorithm is the first contribution of the paper.*

Interestingly, while our work was originally targeted to static analysis of BP specification, it turns out that our choice of ranking metrics, and the corresponding query evaluation technique, are also valuable for the analysis of run-time processes. An *instance* of a BP specification is an actual running process that follows the logic described in the specification. BP Management Systems allow to trace instance executions and record the invoked activities. The execution traces recorded are typically selective and contain only partial information on the activities that were performed at run-time [6]. This may be due to performance concerns, lack of storage space, confidentiality, etc. However, given such a partial log, users often wish to nevertheless understand what had actually happened at run-time. In other words, they would like to identify the execution flows that are most likely to have been the *origin* of this log, typically focusing of specific parts of the log that are of particular interest [17, 4]. Intuitively, the recorded sub-flow may be viewed as a query, with the query result being the set of (relevant parts of) flows that are most likely to have occurred in practice. We show that our query evaluation algorithms can be adapted to retrieve the most likely origins of the given (sub)trace. We first consider the case where the names of the BP activities that are masked/omitted from the trace are known in advance, then the more difficult case where this information is not given. A challenge here is to avoid considering all possible sets of potential omitted/masked operation names (whose number is exponential in the size of the BP). To overcome this we prove a *small world* theorem, showing that only a polynomial number of representative options need to be tested. *This algorithm for computing the top-k possible trace origins is the second contribution of the paper.*

The model that we use here for modeling BP specifications and flows likelihood is borrowed from [14] and is natural abstraction of the BPEL standard (Business Process Execution Language [5]) for BP specifications. A BP is modeled as a nested DAG consisting of activities (nodes), and links (edges) between them, that detail the execution order of the activities. The DAG shape allows to describe *parallel* computations. Activities may be either atomic, or compound. In the latter case their possible internal structure (called implementations) are also detailed as DAGs, leading to the nested structure. A compound activity may have different possible implementations, corresponding to different user choices, variable values, servers availability, etc. These are captured by logical formulas (over the user choices, variable values, etc.) that *guard* each of the possible implementations. In practice, some user choices/variable assignments/server states (and consequently truth values of guarding formulas/implementation choices) are more likely than others. This is captured by a c–likelihood (for *choice likelihood*) function that describes the probability of each implementation choice. This c–likelihood function is then used to define a second likelihood function, this time over possible execution flows, denoted f–likelihood (*flow likelihood*). To simplify the presentation we first assume independence between implementation choices and present our algorithms in this setting. Then we withdraw this assumption and study the implications. In particular we show that computation is still possible (although becomes EXPTIME) if only a bounded level of dependency (to be defined formally in the paper) is allowed. *This top-k query evaluation in the presence of (bounded) dependencies is the third contribution of this paper.*

Paper organization. In Section 2 we formally define our model and query language and the notion of (top-k) projection queries. In Section 3 we present our algorithm for computing top-k query answers, assuming c–likelihood independence. Applications of our technique in the setting of partial execution traces is considered in Section 4, and dependencies between c–likelihood values are introduced in Section 5. Section 6 provides an overview of related work. We conclude in Section 7.

2. PRELIMINARIES

We start by presenting the formal definitions for our model and query langauge.

BP specification. As a running example, we consider a web-based travel agency. The user, searching for a trip, may choose between searching only for flights and searching for combined deals of flights & hotels or flights & hotels & cars. Advertisements are injected in parallel to the search. After viewing the search results, the user may select her reservations. Then, she can either *confirm* her selection, *reset* and re-start the search, or *cancel* and leave the website without finalizing any reservation.

The business logic of the travel agency BP is described schematically in Figure 1. BP specifications are modeled as *nested Directed Acyclic Graphs (DAG)*. These are sets of node-labeled DAGs, each intuitively corresponding to a specific function or service. The graphs consist of *activities* and the flow thereof. Each activity is represented by a pair of *nodes*, the first (having darker background) standing as the activity's *activation point* and the second as its *completion point*. These notions of activation and completion points will be utilized when considering execution flows of a BP. *Edges* represent *execution flow* relations; multiple edges going out of a single node stand for *parallelism* (hence the DAG structure).

Activities may either be *atomic* (like the *Login* activity) or *compound* (like $chooseTravel$ and $Flights$), in the latter case their possible internal flows (called *implementation*) are also detailed (depicted as bubbles). A compound activity may have different possible implementations, corresponding to different user choices, variable values, etc. These are captured by logical formulas (over the user choices, variable values, etc.) "guarding" each possible implementation. The $chooseTravel$ activity has three possible implementations

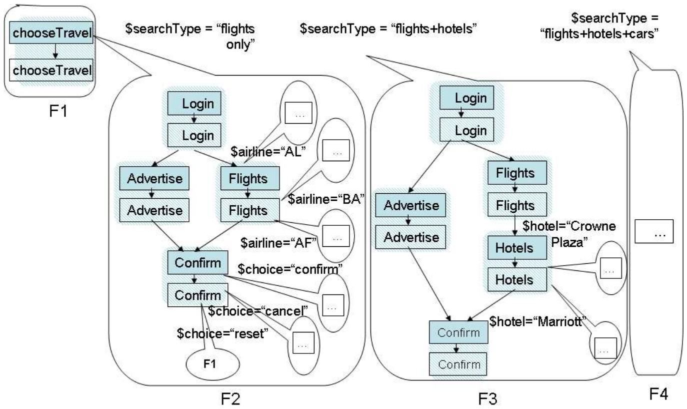

Figure 1: Business Process

$F2$, $F3$, $F4$, and their guarding formulas test the value of the $\$searchType$ variable. This value may either be ”flights only”, ”flights+hotels”, or ”flights+hotels+cars”, depending on the user’s choice. At run-time, exactly one implementation will be chosen, determined by the truth value of the guarding formulas (determined, in turn, by the user choice). Focusing on the $F2$ implementation, the $Flights$ activity has a set of possible implementations, corresponding to choice of $\$airline$ (“BA” stands for British airways, “AF” for Air France, “AL” for Aer Lingus). Last, the $Confirm$ activity has, as one of its implementations, the option of *reset*, recursively going back to $F1$. Other options here are confirmation and cancelation.

To formally define BP specifications, we assume the existence of three domains, $\mathcal{N}$ of nodes, $\mathcal{A}$ of activity names, and $\mathcal{F}$ of guarding formulas[1] . We distinguish two disjoint subsets of $\mathcal{A}$, $\mathcal{A} = \mathcal{A}_{atomic} \cup \mathcal{A}_{compound}$, representing atomic and compound activities, resp. We also use two distinguished symbols `act`, `com`, denoting, resp., activity activation and completion. We first define the auxiliary notion of *activation-completion labeled DAGs*, then use it to define BP specifications.

Definition 2.1. *An* activities DAG *is a tuple* (N, E, λ) *in which* $N \subset \mathcal{N}$ *is a finite set of nodes,* E *is a set of directed edges with endpoints in* N*,* $\lambda : N \to \mathcal{A}$ *is a labeling function for the nodes, labeling each node by an activity name. The graph is required to be acyclic. An* activation-completion *(act-comp) DAG* g *is obtained from an activities DAG by replacing each node* n*, labelled by some label* a*, by a pair of nodes,* n', n''*, labelled (resp.) by* $(a, \texttt{act})$ *and* $(a, \texttt{com})$*. All incoming edges of* n *are directed to* n' *and all of the outgoing edges of* n *now outgo* n''*. A single edge connects* n' *to* n''*.*

We assume g *has a single* **start** *node without incoming edges, and a single* **end** *node without outgoing edges, denoted by* $\texttt{start}(g)$ *and* $\texttt{end}(g)$*, resp.*

Next, we define the notion of BP specifications.

Definition 2.2. *A* BP specification s *is a triple* (S, s_0, τ)*, where* S *is a finite set of act-comp DAGs,* $s_0 \in S$ *is a distinguished DAG consisting of a single activity pair, called the* root*,* $\tau : \mathcal{A}_{compound} \to 2^{\mathcal{F} \times S}$ *is the* implementation function*, mapping each compound activity name in* S *to a set of pairs, each containing a logical formula in* $\mathcal{F}$*, called a* guarding formula*, and an act-comp DAG from* S*.*

At run-time, exactly one implementation is chosen for each occurrence of a compound activity. Its corresponding guarding formula is said to be satisfied*, and we assume that no two guarding formulas (guarding implementations) of the same activity, may be satisfied concurrently.*

For example, for the BP depicted in Figure 1, the set s of act-comp DAGs consists of $\{F1, F2, F3, F4\}$, $s_0 = F1$ is the specification root, and the implementation function τ is depicted by the “bubbles”, annotated by guarding formulas.

Execution Flows. An execution flow is an actual running instance of a Business Process. It may be abstractly viewed as a nested DAG, containing node-pairs that represent the activation and completion of activities, and edges that represent flow and zoom-in (implementation) relationships among activities, along with a record of guarding formulas corresponding to chosen implementations of compound activity nodes.

Example 2.1. *Fig. 2 depicts two example execution flows of the travel agency process. Focus for instance on Fig. 2(a) in which the user chooses a “flights only” search, followed by a choice of British Airways flight, and then confirms her choice. Flow edges are marked by regular arrows, while zoom-in edges, connecting activation and completion nodes of compound activities to the start and end (resp.) nodes of the chosen implementation, are marked by dashed arrows. Fig. 2 (b) depicts a different possible flow, in which the user chooses a “flights+hotels” search, reserving a “British Airways” flight and a “Marriott” hotel. The* f_i*’s next to some nodes are identifiers, considered below.*

Next, we formally define the notion of execution flows for a given BP specification.

Definition 2.3. *Given a BP specification* $s = (S, s_0, \tau)$*,* g *is an execution flow (EX-flow) of* s *if:*

- g *consists only of the root activity pair* s_0 *of* s*, or,*
- *if* g' *is an execution flow of* s*, and* g *is obtained from* g' *by attaching to an activity pair* (n_1, n_2) *of* g'*, labeled by some compound activity name* a*, and having no zoom-in edge attached to it, some implementation* g_a *of the activity* a*, through two new edges, called* zoom-in edges $(n_1, \texttt{start}(g_a))$ *and* $(\texttt{end}(g_a), n_2)$*, and annotating the pair with the formula* f_a *guarding* g_a*. We call* g *an* expansion *of* g'*, and denote* $g' \to g$*.*

g *is called a* full flow *if it cannot be further expanded (that is, it contains the internal flow for each compound activity node*

[1] Formulas are defined in a standard manner using predicates (e.g. (in)equality,<,>) on variables values and connectors (*and*, *or*,*not*).

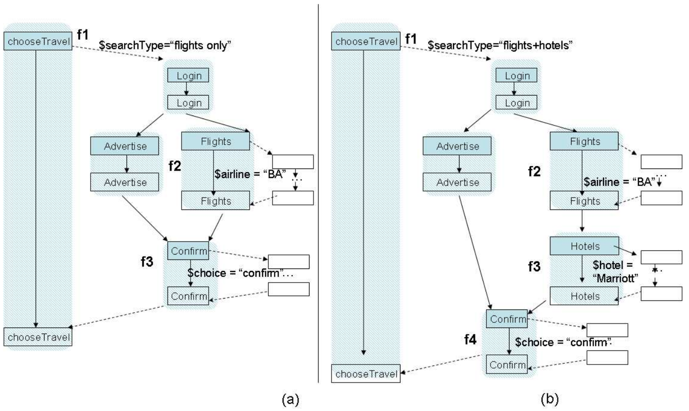

Figure 2: Execution Flow

in it), and a partial *flow otherwise. The set of all full* EX-flows *defined by a BP s is denoted $flows(S)$. For a graph e, we say that e is an EX-flow if a (partial or full) flow of some BP specification S.*

Likelihood. Some user choices / variable values are more common than others, and thus execution flows vary in their likelihood of occurring in practice. We define two kinds of likelihood functions. The first, named c–likelihood (choice likelihood) function, associates a value with each guarding formula (implementation choice). This value stands for the probability that the formula holds. For now we assume *independence* between holding formulas / implementation choices made. We withdraw this assumption later on.

Using the c–likelihood function we may define a second likelihood function, this time over the possible execution flows, namely f–likelihood : recall that each execution flow corresponds to a sequence of implementation choices made during the execution; the joint likelihood of their guarding formulas thus stands as the flow likelihood. Formally,

Definition 2.4. *Given a BP s rooted at s_0 and a c–likelihood function δ for its guarding formulas, the* f–likelihood Δ *of an EX-flow e of s (w.r.t. δ) is defined as follows:*

1. *If e is an EX-flow consisting only of the root activity s_0, $\Delta(e) = 1$.*
2. *Else, if $e' \to e$ for some EX-flow e' of s, then $\Delta(e) = \Delta(e') \times \delta(f)$, where f is the formula guarding the implementation that is added to e' to form e.*

Note that the c–likelihood function δ considered here implies *independency* between choices; that is, the likelihood of any formula to hold is constant, regardless of its context in the flow and truth values of other formulas / implementation choices taken. In general, choices may be dependent: for instance, the choice of hotel may be dependent on the choice of airline that preceded it, etc. In Section 5 we discuss extensions of our results to a more general context where choices are dependent.

$\$searchType$	$P(\$searchType)$	$\$airline$	$P(\$airline)$
"flights only"	0.5	"BA"	0.7
"flights+hotels"	0.25	"AF"	0.2
"flights+hotels+cars"	0.25	"AL"	0.1
$\$hotel$	$P(\$hotel)$	$\$choice$	$P(\$choice)$
"Marriott"	0.6	"reset"	0.6
"HolidayInn"	0.3	"confirm"	0.4
"CrownePlaza"	0.1		

Table 1: c–likelihood function

Example 2.2. *Consider Table 1 that depicts the probability of value assignments for the different variables of the travel agency BP, and consequently the c–likelihood function for the corresponding guarding formulas. According to this c–likelihood function, the f–likelihood of each EX-flow may be computed. For instance, the f–likelihood of the EX-flow in Fig.2(a) is computed as the multiplication of the c–likelihood values of \$searchType = "flights only", \$Airline = "British Airways", and \$choice = "confirm", that is $0.5 * 0.7 * 0.4 = 0.14$. The f–likelihood of the EX-flow in Fig.2(b) is computed as the multiplication of $\$searchType =$ "$flights + hotels$", $\$airline =$ "$BritishAirways$", $\$hotel =$ "$Marriott$", and $\$choice =$ "$confirm$", i.e. $0.25 * 0.7 * 0.6 * 0.4 = 0.0672$.*

Queries. We distinguish between *selection* and *projection* queries. The former seeks for full execution flows that contain some given execution pattern, while the latter focuses only on sub-flows that are of interest to the user. We start by defining the auxiliary notions of *execution patterns* and their *embeddings* within execution flows. Then, we utilize these definitions to define the syntax and semantics of selection and projection queries.

Execution patterns are adaptations of the tree- and graph-patterns offered by existing query languages for XML and graph-shaped data, to nested EX-flow DAGs. Namely, such patterns are similar in structure to EX-flows, but contain *transitive edges* that may match any execution flow *path*, and *transitive nodes*, for searching within indirect implementations, at any level, of the corresponding composite node. Nodes in the pattern may be labeled by the wildcard ANY, that may match any EX-flow node.

Definition 2.3. *An* execution pattern, *abbr. EX-pattern, is a pair $p = (\hat{e}, T)$ where $\hat{e}$ is an EX-flow whose nodes are labeled by labels from $\mathcal{A} \cup \{\text{ANY}\}$ and may be annotated by a formula. T is a distinguished set of activity pairs and edges in $\hat{e}$, called* transitive *activities and edges, resp.*

Example 2.4. *Figure 3(a) depicts a simple execution pattern describing flows that contain some "British Airways" (Abbr. "BA") flight search that resulted in a confirmation (ignore for now the rectangle surrounding a sub-graph of the pattern). The double-lined edges are* transitive edges. *The doubly bounded $chooseTravel$ nodes are* transitive nodes. *The q_i labels next to some nodes are identifiers, and will be utilized below.*

Intuitively, the transitive edge connected to the $chooseTravel$ may match any sequence of searches and resets, and the transitive edge connecting the $Flights$ and $Confirm$ activities may match any sequence of search activities (for hotels

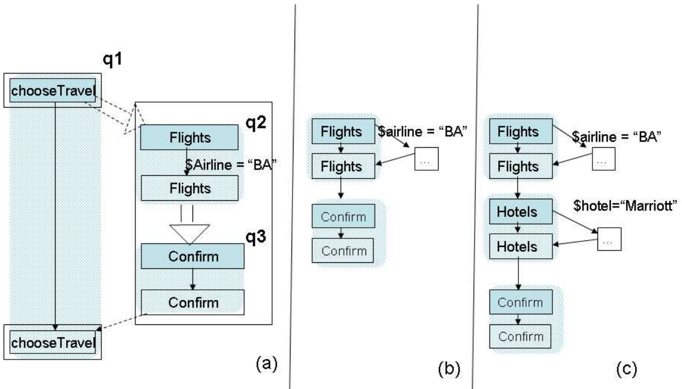

Figure 3: Query And Answer

and/or cars). The transitivity of the chooseTravel node allows this matching to include indirect implementation of the corresponding composite node, at any nesting level.

We next formally define the notion of *embeddings*, which are matchings of execution patterns to execution flows.

Definition 2.5. *Let $p = (\hat{e}, T)$ be an execution pattern and let e be an EX-flow. An* embedding *of p into e is a homomorphism ψ from the nodes and edges in p to nodes, edges and paths in e s.t.*

1. **[nodes]** *activity pairs in p are mapped to activity pairs in e. Node labels and formulas are preserved; a node labeled by* ANY *may be mapped to nodes with any activity name.*
2. **[edges]** *each (transitive) edge from node m to node n in p is mapped to an edge (path) from $\psi(m)$ to $\psi(n)$ in e. If the edge $[n, m]$ belongs to a direct internal trace of a* transitive *activity, the edge (edges on the path) from $\psi(m)$ to $\psi(n)$ can be of any type (flow, or zoom-in) and otherwise must have the same type as $[n, m]$.*

We next define queries. We consider two types of queries. A *selection query* simply consists of an EX-pattern. When applied on a BP specification it selects all EX-flows in which the pattern may be embedded. *Projection queries*, on the other hand, allows to focus on some specified part of the selected flows. Formally,

Definition 2.6 (Selection Queries). *Given an execution pattern $p = (\hat{e}, T)$ and a BP specification s (along with a c–likelihood function δ) we define the result of evaluating p over s, denoted by $p(s)$, as the set of all execution flows e of s such that there exists an embedding of p within e. $top - k(q, s)$ is a set of k execution flows of $p(s)$ having highest f–likelihood values* [2]*.*

Projection queries consists of an execution pattern along with a sub-graph annotated as output. Only nodes and edges matching this part are projected out and appear in the query result.

Definition 2.7 (Projection Queries). *A projection query $q = (p, P)$ consists of an execution pattern p accompanied by a sub-graph P of the pattern, itself forming an execution pattern, called the* projected *part of the pattern.*

- *For an EX-flow e and an embedding h of p in e, the result of q on e, w.r.t. h, is denoted $q_{\downarrow}(e, h)$ and contains all nodes and edges of e to which nodes and edges of P are mapped by h. For each activity pair participating in the embedding, the edge connecting its activation and completion nodes is also included.*
- *For an EX-flow e, the result of q on e, denoted $q_{\downarrow}(e)$, consists of the results of all possible embeddings of q into e, i.e. $q_{\downarrow}(e) = \bigcup_h q_{\downarrow}(e, h)$.*
- *Finally, for a BP s, the result of q on s, denoted $q_{\downarrow}(s)$ is the set of all possible results of q when applied on the EX-flows of s. Namely $q_{\downarrow}(s) = \bigcup_{e \in flows(s)} q_{\downarrow}(e)$.*

It is easy to prove that $q_{\downarrow}(s)$ is a set of EX-flows. Note that an EX-flow $e' \in q_{\downarrow}(s)$ may originate from several EX-flows of s, namely there may be several $e \in flows(s)$ s.t. $e' \in q_{\downarrow}(e)$. The *score* of e' is the maximum likelihood of these flows, namely $score(e') = max\{\Delta(e) \mid e \in flows(s) \wedge e' \in q_{\downarrow}(e)\}$, where Δ is the f–likelihood function for s.

Definition 2.8. *The top-k result of a projection query q over a BP specification s, denoted $top - k(q_{\downarrow}, s)$, is a set of EX-flows in $q_{\downarrow}(s)$ having highest score values.*

Example 2.9. *Re-considering the pattern of Fig. 3(a) and the EX-flow of Fig. 2(a), a possible embedding h maps q_1 to f_1, q_2 to f_2, and q_3 to f_3. The activation and completion nodes of f_1, f_2, f_3 thus participate in the projection result, along with their connecting edges. The transitive edge from q_1 to q_2 is mapped to the path from f_1 to f_2 containing the zoom-in edge, the Login activity and the edge from Login to Flights. The transitive edge from q2 to q3 is then mapped to the implementation of Flights, along with the zoom-in edges, and the edge connecting to Hotels. The projection result is depicted in Fig. 3(b).*

Embedding the pattern in the EX-flow of Fig. 2(b) is similarly done, and the projection result is given in Fig. 3(c).

Evaluating the projection query on the BP specification of Fig. 1, the top-1 projection is depicted in Fig. 3(b), with the maximal likelihood leading to it being the flow of Fig. 2(a), with likelihood of 0.14 (which is thus the projection score).

Technical Remark. We have required in Definition 2.2 above that no two guarding formulas of the same activity may be satisfied concurrently. Consequently, the sum of likelihoods of guarding formulas for each activity name was 1. While this assumption complies with an intuitive concept of BP specification, they are not necessary for our results. Indeed, we use below, as a tool, relaxed BPs without this requirement.

[2] Since distinct EX-flows may have the same likelihood this set may not be unique and we choose one arbitrarily

3. QUERY EVALUATION

We describe in this section our evaluation algorithm for top-k projection queries. We first recall the algorithm for evaluating top-k selection queries, given in [14]. We show that direct adaptations of this algorithm to projection queries fail at achieving PTIME query evaluation, and finally describe an alternative algorithm that overcomes this.

3.1 Evaluating Selection Queries

A main tool in the efficient evaluation of top-k (selection and projection) queries over a given BP, is a compact representation of (intermediate) evaluation results via another BP specification. Thus, we start by defining when a BP specification s *captures* a set of EX-flows along with their likelihoods.

DEFINITION 3.1. *A BP s' along with a c–likelihood function δ' (corresponding f–likelihood function Δ')* captures *a (possibly infinite) set of EX-flows T, with respect to a f–likelihood function Δ over T if (1) $flows(s')$ is identical* [3]*, up to some renaming function ρ to its activity names, to T and (2) $\forall_{e \in flows(s')} \Delta'(e) = \Delta(\rho(e))$.*

Given a BP specification s and a query q, the *TOP-K-SELECTIONS* algorithm, given in [14], finds a BP s'' along with δ'' that captures $top-k(q, s)$. The algorithm operates in two steps:

- *AllFlows*: first, the algorithm finds a BP specification s', δ' that captures *all flows* corresponding to the query (i.e. $q(s)$).
- *Top-k*: second, the algorithm analyzes the obtained representation s' to find a representation s'', δ'' of only the top-k most likely flows of s'. s'', δ'' thus captures top-k(q, s).

The algorithm bears two important features: first, the execution time incurred by the construction of a BP that captures top-k(q, s) is *polynomial* in the size of the specification; second, explicit construction of any concrete flow out of the flows in top-k(q, s) is possible in time which is polynomial in the output size (i.e. the size of the flow). Our goal is to design an evaluation algorithm for *projection* queries bearing these two features.

The details of the above algorithm may be found in [14], but are not important here, as we next show that its possible direct adaptations to the setting of projection queries are generally infeasible.

3.2 Adaptations of TOP-K-SELECTIONS

We next present two possible adaptations of TOP-K-SELECTIONS for projection queries, and show that they may incur exponential data complexity.

First attempt - Explicit Enumeration of EX-flows. The first simple approach for evaluating projection queries is to try and use the above mentioned algorithm that constructs a compact representation s'', δ'' of top-k(q, s) [4], and then to explicitly enumerate the flows of s', along with their f–likelihood values. Next, for each obtained flow, we may compute its corresponding projections. Each projection result is ranked based on the maximal f–likelihood of a flow leading to it.

Note, however, that this naive algorithm may incur time that is exponential in the **BP specification** size, as well as in the *output size*, as the following theorem holds:

THEOREM 3.2. *There exists a fixed-sized query q such that for every n there exists a BP specification s_n whose size is linear in n, such that even the smallest size of any EX-flow of $q(s_n)$ is exponential in n, while the size of the largest $q_{\downarrow}(s_n)$ is bounded by a constant.*

PROOF. Consider a BP specification in which each activity has a single implementation, as follows: the implementation of each activity a_i $(i = 1, ..., n-1)$ consists of two nodes labeled a_{i+1}, and the implementation of a_n consists of a single atomic activity. Consider also a query that seeks for the root activity, whose (indirect) implementation consists of two *any*-labeled nodes, connected by a transitive edge. The query then projects over the root node. It is easy to observe that $q(s_n) = s_n$, and it has only a single possible EX-flow. The size of this flow (containing two instances of a_2, 4 instances of a_3, etc.) is **exponential** in n. Note that in contrast, $q_{\downarrow}(s_n)$ consists of a single activity pair. □

Second attempt - Computing a compact representation of all projections. An alternative approach for adapting the selection queries evaluation algorithm follows the idea of generating a compact representation of all results, namely $q_{\downarrow}(s)$, and then retrieving the top-k out of these. Unfortunately, this approach would also lead to infeasible algorithm, as the following theorem holds:

THEOREM 3.3. *There exists a fixed-sized query q such that for every n there exists a BP specification s_n whose size is linear in n, such that* **no** *s'_n whose size is polynomial in the size of s_n captures $q_{\downarrow}(s_n)$.*

PROOF. For a number n, consider a BP specification s_n, in which the start node branches into two different activity nodes, then joins with edges going into a single node, then branches again, and so on, n times. Now consider a query q that projects over the path from the start to the end node. For a BP specification s'_n to capture $q_{\downarrow}(s_n)$, each possible path in s_n must appear as a direct implementation of the root activity in a distinct EX-flow of s'_n. The number of such paths in s_n is 2^n, and thus the root activity of s'_n must bear at least as much direct implementations. As a consequence, the size of s'_n is exponential in the size of s_n (the size of the latter is linear in n). □

[3] We use isomorphism for equivalence between graphs

[4] If multiple flows lead to the same projection result one may need to generate the top-k' flows for some $k' > k$.

Next, we show an alternative, PTIME, algorithm for evaluating top-k projection queries.

3.3 Efficient Evaluation of Top-k Projection Queries

As shown above, it is infeasible to compute a compact representation for *all* projection results and then retrieve the top-k ones out of it. However, we may still perform a two-steps algorithm, in the spirit of TOP-K-SELECTIONS. The second step (*top-k*) stays intact. However, instead of a first step that generates a specification capturing the *entire* set of results $q_\downarrow(s)$, the first step of the refined evaluation algorithm generates a specification that captures only a *subset* of $q_\downarrow(s)$, including in particular the top-k ranked projections. We say that such BP *k-captures* $q_\downarrow(s)$. Formally,

DEFINITION 3.4. *Given two BP specifications s', s, a query q, and a number k, we say that s'* k-captures *$q_\downarrow(s)$ if $top-k(q_\downarrow, s) \subseteq flows(s') \subseteq q_\downarrow(s)$ (up to some renaming function over the nodes), and the score of each projection f is the same as the f–likelihood of the corresponding flow of s'.*

THEOREM 3.5. *Given a BP specification s and a projection query $q = (p, P)$, we may compute a BP specification s' that* k-captures *$q_\downarrow(s)$ in time polynomial in s, with the exponent determined by the query size.*

PROOF. We present an evaluation algorithm, K-CAPTURES, computing s' that *k-captures* $q_\downarrow(s)$. Throughout the computation, our algorithm uses an EMBED function, to embed (sub-)patterns into sub-graphs of implementations. When invoked on a graph g and a pattern p, EMBED returns a set of graphs $em(g, p)$, each corresponding to a single embedding: each $g' \in em(g, p)$ is isomorphic up to activities names to some sub-graph of g. The activity labeling each node encodes the identifier and label of the original node in g, as well as of all nodes (transitive edges) of p mapped to it. EMBED may be implemented using extensions of sub-graph homomorphism algorithms [9].

As the algorithm details are intricate, we explain it in 4 steps, gradually considering further complicated patterns.

"Flat" Patterns. We start by considering simple execution patterns that consist of a root activity whose implementation contains only atomic activity nodes, or compound activity nodes having no implementation attached to them. Additionally, the pattern contains no transitive nodes or edges.

The algorithm begins by embedding the pattern root r' in the BP root r, creating a new root $[r, r']$. Then, we use EMBED to try embedding the implementation of r' in each direct implementation of r. Consider an implementation g of r, guarded by f, s.t. $em(g, p) \neq \phi$ and let $g' \in em(g, p)$. g' is an implementation of the newly created $[r, r']$, and its guarding formula is denoted by $f_{[r,r']}$.

We then compute the c–likelihood value δ' assigned to $f_{[r,r']}$, as follows. Denote by N the set of all compound activity nodes that appear in g. The implementations of N were not matched and do not have counterparts in g', as p contains no implementation of any node other than the root. In contrast, recall that the score for each projection is computed as the maximal likelihood of a *flow* leading to it. The most likely such flow contains, as sub-flows, the top-1 implementation for each node in N. We thus need to compute the partial f–likelihood induced by the top-1 implementation of each $n \in N$, denoted by top-1-rank(n), then compensate for it by setting $\delta'(f_{[r,r']}) = \delta(f) * \Pi_{n \in N}$top-1-rank(n). To compute the top-1-rank(n) we employ the Top-k algorithm (described in Section 3.1, in the second step of the TOP-K-SELECTIONS algorithm), with $k = 1$ and n being the BP root activity.

Nested Patterns. We next consider execution patterns in which some nodes in the root's implementation may have implementations attached to them by zoom-in edges (and nodes in these implementations may bear implementations, and so on).

In this case, to proceed with the embedding, and for each matched compound activity of the pattern that bears an implementation, we then match recursively its implementation (in the pattern), to any implementation (in the BP) of its matching node's activity. These recursive calls end when either the pattern contains no more nodes (in this case a successful match was found), or when the pattern contains nodes, but no embedding is found for them. In the latter case we mark this matching as a failure. A "garbage collection" step follows, removing each activity for which all implementations lead to failure (then activities that only lead to these activities, etc.).

As for computation of f–likelihood values, implementations of some of the compound nodes of g were matched, and are considered in further computation. Thus we only need to compensate for those compound nodes that were not matched; consequently, the set N contains only these nodes.

Patterns with transitive nodes. If the query pattern includes transitive nodes, their implementations may need to be embedded in indirect implementations of the corresponding BP nodes. Consequently we must consider all possible *splits* of the patterns that appear as implementations of transitive nodes, into smaller sub-patterns. For each such split of a pattern p into sub-patterns $p_1, ..., p_k$, we embed p_1 within a direct implementation I of the corresponding compound node; as for $p_2, ..., p_k$, we "assign" each to a compound node of I, generating a node with a new activity name $[a, p_i]$, where a is the node's original activity name and p_i is the assigned sub-pattern. Intuitively, $[a, p_i]$ "guarantees" the embedding of p_i in its indirect implementation. We then recursively construct the embedding of p_i in an implementation of a. A "garbage collection" step removes implementations leading to failure.

The algorithm's termination is guaranteed as each sub-pattern of the query is matched only once against implementations of each specification node. If a match is found, the generated activity name guarantees the existence of a match, and it does not need to be further verified.

Computation of f–likelihood values is again refined, accounting for the case where upon matching a transitive node, an entire implementation graph I of some node n is omitted, while implementations of node within it are kept intact. Thus, the "compensation" of likelihood is generalized: the c–likelihood of each newly generated formula guarding an implementation g'', is multiplied by the top-1-rank of all nodes co-appearing with nodes whose implementation generated a sub-graph of g'', in implementations that were eliminated.

Patterns with transitive edges. Finally, we consider transitive edges. The end-nodes of each edge are matched as above to specification nodes (denote these as n for the start node of the edge and m for the end node), and it remains to consider the paths in-between them. If the transitive edge does not appear in the projection part, we only need to verify that a path exists. When the transitive edge does appear in the projection part, then each projection result contains a path, to which the edge is matched. It is infeasible to create an implementation for each distinct projection (i.e. each distinct path), as the number of paths between two nodes of the specification may be exponential in the specification size. Fortunately, we are only interested in the top-k projections, in which only the top-k paths may appear.

We also assume in the sequel that the transitive edge appears within an implementation of a transitive node. If not, then any k [5] paths may be chosen, as they all co-appear in the same implementation, thus share the same likelihood.

We design a Dynamic Programming algorithm, namely TOP-K-PATHS, generating the top-k paths from n to m.

TOP-K-PATHS. First, we assign a unique identifier to each specification node, then generate a new set of nodes $[n_j, i]$ for each n_j of the specification and for each $i = 1, .., k$, and initialize a table which keeps track of the i'th most likely path from each n_j to m, along with its likelihood. All entries are initialized to an empty path and zero likelihood, except for the entry for m itself, assigned a likelihood of 1. When the table is full, it contains in particular the top-k paths from n to m, which are explicitly generated.

To fill in the table, we use the auxiliary notion of *children*, as follows. We say that n_2 is a child of n_1 if there is an edge from n_1 to n_2, or if n_1 is a activation node of a compound activity and n_2 is a start node of its implementation, or if n_2 is an completion node of a compound activity and n_1 is an end-node of its implementation.

The computation of values in the table proceeds as follows, computing the i'th paths for increasing values of i. When computing the top-i path originating at some node l, we consider all *children* of l in the specification. For each such child u, say that j is the greatest index of a path originating in u that was used as a sub-path in some $r < i$ ranked path of l that was previously computed. Then we compute the $j + 1$ path originated in u (if it was'nt already computed), and obtain $score(u)$, the score of u as a candidate for the path being generated for l. In case the edge between l and u is a zoom-in edge, $score(u)$ is multiplied by the c–likelihood of f, where f is the formula guarding the implementation rooted at u. The child of u with the maximal score, along with its corresponding sub-path, is chosen for the top-i path originating at l.

Each obtained path serves as a separate implementation of the corresponding compound activity. The following lemma holds.

LEMMA 3.6. *The partial specification generated by TOP-K-PATHS algorithm captures the k most likely paths from n to m.*

Complexity. The complexity of the above algorithm is polynomial in the BP size s, with the exponent determined by the query size q. The polynomial data complexity is ensured by the polynomial data complexity of EMBED, as well as by our treatment of transitive nodes (where each specification node is matched only once against each query sub-pattern) and of transitive edges (generating only k paths rather than all paths).

□

COROLLARY 3.7. *We may compute a BP specification s'' that* captures *$top - k(q_\downarrow, s)$ in PTIME (data complexity).*

PROOF. The TOP-K-PROJECTIONS algorithm first applies K-CAPTURES, to obtain s' that k-captures $q_\downarrow(s)$. s' flows capture all results of $top-k(q_\downarrow, s)$, but possibly some additional flows. Thus, we apply the TOP-K algorithm, which served as step (2) of the TOP-K-SELECTIONS algorithm above. TOP-K is a dynamic programming algorithm, similar to TOP-K-PATHS. Its details may be found in [14] and are thus omitted here.

□

Note that the exponential dependency on the query size is inevitable unless $P \neq NP$, as the following theorem holds.

THEOREM 3.8. *Given a BP specification s , a query q and a threshold t it is NP-complete with respect to the query size to decide whether there exists a projection $q_\downarrow(e)$ with score greater than t.*

NP-hardness may be proved by reduction from emptiness of the results set, shown in [12] to be NP-hard. The NP algorithm is based on a "Pumping Lemma", guaranteeing that it suffices to "guess" a small (polynomial) sized flow, then project it (details omitted for space constraints). Interestingly, the problem remains NP-hard even when all specification and query graphs have a simple, almost tree-shaped form.

[5] If there are only $k' < k$ paths, then all are matched

4. EMPLOYING SELECTIVE TRACING

We have studied above projection queries as a tool for analysis of Business Process specifications. In this section we consider another application of our results. It is a common practice to *trace* process executions. A *naive* tracing keeps exact record of the activities occurring along the flow, and the obtained traces are thus equivalent, in this case, to execution flows. However, typical tracing is not naive. First, the exact names of some activities are sometimes not recorded. For instance, a generic name, say "Payment" may be used to record a set of payment-related activities (e.g. cash payment, credit card payment). Second, some activities may be omitted from traces altogether, due to lack of storage space, confidentiality, lack of interest, etc.

Given such a partial trace, the user may wish to identify flows that are most likely to have occurred in practice, and moreover, focus on parts of these flows that are of interest. We first describe our model for partial traces, then study identification of most likely flows, given a partial trace.

4.1 Modeling Partial Traces

The omittance of activities and formulas from execution traces, as well as the renaming of other activities, is captured through the notion of *selective tracing systems.* Formally, a selective tracing system $\sigma = (A, F, \pi)$ for a BP specification s consists of a set A (F) of activity names (guarding formulas) in s to be omitted, and a renaming function π from activities names in s to activities names in $\mathcal{A}$. The obtained traces are called *selective traces*, and to define them we recall the definition of [13] [6]:

Definition 4.1. **[Selective EX-traces]** *Given a BP specification s and a* selective tracing system $\sigma = (A, F, \pi)$ *satisfying condition (*) below, the set of* selective traces *defined by s and σ, denoted $traces(s, \sigma)$, consists of all graphs e obtained from EX-flows of s by deleting all activity pairs with labels in A,* [7]*, deleting all occurrences of guarding formulas in F, and then replacing each label a of the remaining nodes by $\pi(a)$.*

Condition (*): *A does not include the root activity of s, and for each activation-completion graph g in s, the graph g' obtained from g by removing the atomic activity pairs with names in A is itself an activation-completion graph (as in Definition 2.1), or is empty.*

The intuition behind condition (*) is that from a practical point of view, it is reasonable to assume that the result obtained after each loss of information still bears the shape of a flow.

Consider, for example, a selective tracing system (A = Hotel, F = \$*searchType* = "flights only", \$*searchType* = "flights+hotels"), with π being the identity function. Under such tracing, it is impossible to distinct between a flow containing a search for airlines and hotels and one containing a search for only airlines, as all occurrences of the *Hotels* activity, and of formulas dictating the search type are omitted. Indeed, the same trace is obtained for the two different flows of Fig. 2.

We next define the notion of *origin* flows, which are flows that may have occurred given an observed trace, as follows:

Definition 4.1. *An execution flow e is a possible* origin *of an execution trace t, w.r.t. a selective tracing system $\sigma = (A, F, \pi)$, denoted $\sigma(e) = t$, if deleting from e all occurrences of activity names in A and of formula names in F, and renaming all other activities according to π, results in t.*

Given an execution trace t we wish to identify the top-k flows that are likely to have occurred in practice, or relevant parts thereof. We consider a more generalized setting, where the input consists of a query, and we seek for (relevant parts of) the ex-flows most likely to occur, out of *the origins of any trace conforming to the query.* (This setting is indeed more general: an observed trace t may be represented by the pattern, with the projection part being the part of interest).

We next define top-k projections of origins, under selective tracing. We say that $e' = e$ ($e' \subseteq e$) if e' is isomorphic to (a sub-graph of) e.

Definition 4.2. *Given a BP specification s, a selective tracing system $\sigma = (A, F, \pi)$ and a query q, we define $origins(q_{\downarrow}, s, \sigma) = \bigcup_{e \in flows(s)} \{e' \subseteq e \mid \sigma(e') = q_{\downarrow}(\sigma(e))\}$, and say that e leads to e'. The score of each projection result e' is defined as the maximal f-likelihood of a flow e leading to it, and $top-k(q_{\downarrow}, s, \sigma)$ consists of the k projections in $origins(q, s, \sigma)$ having the best scores.*

Intuitively, the results set $origins(q_{\downarrow}, s, \sigma)$ contains all sub-flows e' of some flow e of s (before renaming), such that after renaming, $\sigma(e')$ is the projection of $\sigma(e)$ over $q_{\downarrow}$. It thus captures the relevant parts of conforming flows. As before, each sub-flow e' is ranked by the maximal ranking of a corresponding flow e.

In the remainder of the section we discuss computation of $top-k(q_{\downarrow}, s, \sigma)$, in two different settings. First, we assume knowledge of σ, i.e. we know exactly which activities are deleted / renamed. Then, we consider a more intricate scenario where this information is unknown.

4.2 Known Selective Tracing

We start by showing that given q, s, σ, we may efficiently capture $top-k(q_{\downarrow}, s, \sigma)$.

Theorem 4.3. *Given a BP specification s, a selective tracing system σ, and a query q, obtaining a BP specification s' that* captures *$top-k(q_{\downarrow}, s, \sigma)$, may be done in polynomial time (data complexity).*

[6] [13] did not consider formulas, but the adaptation is straightforward.

[7] When an atomic activity pair (n_1, n_2) is deleted, the edges incoming n_1 are now connected to the nodes previously pointed by n_2. For compound activities the incoming(outgoing) edges of n_1 (n_2) are now being connected to the start/end nodes of the implementation sub-graph.

Proof Sketch. Re-visiting the TOP-K-PROJECTIONS algorithm of section 3, we adapt the first step (K-CAPTURES) to the selective setting, as follows:

1. Upon matching of *nodes*, we may match a query node n' to a specification node n if the label of n, *after renaming* is the same as the label of n'.

2. Upon matching of *edges*, we may also match a non-transitive edge to a specification path, given that the path includes only nodes whose labels are in the deletion set of the tracing system, and all zoom-in edges along the path are guarded by formulas in the deletion set. *TOP-K-PATHS* is adapted to support these newly created edges, verifying that nodes and edges (in any nesting level) along the matched path indeed conform to the above requirement.

3. Each composite node is treated as *transitive*, and its corresponding zoom-in edges are treated as transitive, as in item (2).

This concludes the proof. □

4.3 Unknown Selective Tracing

In the previous discussion we have assumed knowledge of the information lost in the tracing process. Such knowledge may generally be absent, for instance if we have no access to the tracing system itself, but only to a sample of traces.

Obviously, assuming no information at all on the tracing system, one cannot infer anything on the flows shape. We thus assume the tracing system follows *some* selective tracing, that is, *all* occurrences of some subset of activity names are deleted, and all occurrences of another subset are renamed.

We then say that an EX-flow is an origin of an EX-trace if it's an origin of it under some selective tracing system, and we correspondingly define top-k-uSelective($q_\downarrow, s$), as follows:

Definition 4.4. *Given a BP specification s and a query q, we define $origins(q_\downarrow, s, uSelective)$ as $\bigcup_\sigma origins(q_\downarrow, s, \sigma)$. $score(e', uSelective) = max_\sigma score(e', \sigma)$, and we denote top-k-uSelective$(q_\downarrow, s)$ as the k results with highest scores out of these in $origins(q_\downarrow, s, uSelective)$.*

The following theorem holds:

Theorem 4.5. *Given a BP specification s, and a query q, we may generate a BP specification s', that* k-captures *$top-k-uSelective(s, q_\downarrow)$, in polynomial time (data complexity).*

Proof. We first give a naive, exponential algorithm that explicitly enumerates all possible tracing systems, then optimize it by considering only a subset of the possible tracing systems. Finally, we prove that it is indeed sufficient to consider only this subset.

Naive Algorithm. Given a BP specification s and a query q, a naive algorithm generates all possible systems $\sigma = (A, F, \pi)$ such that A contains activities names appearing in s ,F contains guarding formulas appearing in s and π is a mapping from activity names in s to activity names in q.

Then, for each such σ, the algorithm finds $s'(\sigma)$ that captures $top-k(q_\downarrow, s, \sigma)$ (possible by Theorem 4.3). Last, the algorithm constructs a BP specification s'' whose root contains as implementations all the $s'(\sigma)$ specifications, and apply a top-k analysis over s'', to find the top-k projections.

As the number of tracing systems tested is exponential in the size of s, such algorithm is infeasible.

Optimized Version. We avoid explicit generation of all possible tracing system, and generate only a subset of these. We first generate all possible mappings from query nodes (formulas) to BP activities names (formula names), ignoring their original activities (formulas). Now, for each such mapping L, we first check L for *consistency*. That is, verify that no two query nodes whose original activities (formula names) were identical, are mapped to different labels (formula names). The algorithm generates a selective tracing system, denoted σ_L, with its deletion set A_L (F_L) including all activity names (formula names) except for those assigned by L to some query node (formula name). The renaming function π_L matches each BP activity name to the original activity name of a query node mapped by L to it (uniquely defined due to L's consistency, all such nodes bear the same activity name). For each consistent mapping L, σ_L is added to a set of tracing systems, denoted Λ. The size of Λ is polynomial in that of s. We claim that Λ captures all tracing systems that need to be checked, namely:

Lemma 4.6. *Given a BP specification s and a query q, and for any e, there exist σ, t s.t. $e \in top-k(q_\downarrow, s)$ if and only if there exist such σ', t', with $\sigma' \in \Lambda$.*

Proof. The first direction is immediate. We thus assume that there exists such σ, t, and denote by L the node embedding of q in t, that is the assignment of labels to query nodes, according to the embedding E of q in t. Denoting the result of applying σ_L on e by t', we construct the embedding E' of q in t': it maps nodes to nodes exactly as E does (note that no node of t to which E maps a *node* of q was removed by σ_L). Edges are mapped to edges connecting corresponding nodes; as for transitive edges, there still exists a path (possibly containing one edge) in t' connecting the corresponding end-nodes, as edges are not removed unless their end-nodes are removed.

□

As the number of tracing systems generated, as well as the evaluation complexity for each tracing system, is polynomial in the BP specification size, we obtain a polynomial data complexity algorithm.

This concludes the proof of Theorem 4.5.

5. DEPENDENCIES

In the above discussion, we have assumed full independency between the likelihood of choices dictating the execution flow. In practice, such independency is rare, as user choices, variables values, etc. tend to be correlated with each other. To conclude, we withdraw the assumption of independency and show that under plausible assumptions, query evaluation is feasible under dependency.

To define dependency, we first extend the definition of the c–likelihood function to consider not only a given guarding formula, but also an EX-flow representing the *history* occurring before the formula evaluation. Namely, δ is now a function of both e' and f, where e' is a partial flow and f is a formula guarding an activity name of a node that is next to be expanded [8]. Intuitively, prediction of the next user choice / variable value is dependent on the history of the flow thus far; for instance, if British Airways and Marriott offer combined deals, we expect to observe that typical British Airways customers stay at the Marriott.

We have shown in [14] that for general c–likelihood functions with dependency, even emptiness of selection queries is undecidable. However, in realistic cases, the likelihood of each formula depends on the execution history, but only in a bounded manner. One way [22] to define such bound is to consider a "sliding window", that is allow dependency of each choice on, at most, the m last choices; however, this approach yields, in our setting, a class that is too restrictive. To observe that, consider a recursive BP, and a choice f_1 (f_2) that precedes (follows) the recursive loop. We would still like to allow dependency between f_1 and f_2, even for executions in which the number of choices within the loop exceeds our bound. Thus, for each formula f, we bound the number of implementation choices previously made for *each activity* a, that affect the likelihood of f. We next define the *bounded-memory* class, capturing such functions, as follows:

Definition 5.1. *Given a BP s, we say that δ is* bounded-memory *if there exists some finite bound n, s.t. for each guarding formula f, for each activity name $a_i \in s$ and for all pairs of EX-flows e, e' that agree on the implementations chosen for each activity name a_i, in its last n expansion steps, $\delta(e, f) = \delta(e', f)$. If $n = 0$, δ is* memory-less.

Continuing with our running example, assume that the choice of hotel is dependent on the last choice of search type, and on the last choice of airline; the memory bound for the hotels activities, in this case, is 1. In contrast, if the likelihood of choosing BritishAirways depends, for example, on exactly how many times "reset" was chosen, then we have an unbounded memory.

We claim that we may extend all of our algorithms to the setting of a bounded-memory likelihood function, though this may incur exponential time in the specification size.

Theorem 5.2. *Given a BP specification s along with a bounded-memory c–likelihood function δ, (a selective tracing system σ), and a query q, generating a BP specification s', that* captures *$top-k(q_\downarrow, s)$ ($top-k(q_\downarrow, s, \sigma)$, $top-k-uSelective(q_\downarrow, s)$) may be done in exponential time (data complexity).*

Proof Sketch. We use the algorithm from [14] that, given a BP specification s and a bounded-memory c–likelihood function δ, generates a BP specification s' and a *memory-less* c–likelihood function δ' that captures $flows(s)$. The activity names of s' encodes sufficient amount of history required for computation of likelihood for guarding formulas, and thus the size of s' may be exponential in the size of s. A renaming function ρ maps the new names into names of s.

The algorithms presented in previous sections may now be applied over s', generating a new s'' capturing $top-k(q_\downarrow, s)$ ($top-k(q_\downarrow, s, \sigma)$, $top-k-uSelective(q_\downarrow, s)$). During this algorithm, a second renaming function ρ' is generated, as explained above. The final renaming function over activity names is obtained as $\rho' \circ \rho$.

□

Unless P=NP, the exponential time of evaluation in presence of dependency may not be improved, as the following theorem holds (proof omitted for space constraints):

Theorem 5.3. *Given a BP specification s, with a bounded-memory c–likelihood function, a query q and a threshold t it is NP-complete with respect to the size of s to decide whether there exists a projection $q_\downarrow(e)$ with score greater than t.*

6. RELATED WORK

We next give a brief review of related work, highlighting our results relative contributions.

The query language and data model that we consider here were originally introduced in [2, 3], and extended to a probabilistic setting in [14]. The work in [14] focused only on *selection queries* and assumed *naive tracing*, i.e. that execution traces accurately and exactly represent the flows that had occurred in practice. In contrast, our work here is the first, to the best of our knowledge, to study *projection queries* and *partial tracing* for Probabilistic Business Processes. The model for partial tracing used here was introduced, for a non-probabilistic setting, in [13]. It was shown to be realistic enough to capture real-life tracing systems such as [6].

Probabilistic XML [1, 20, 19, 8] extends XML to a probabilistic setting, by introducing *distributional nodes*. For such node, a subset of its children is randomly chosen to appear in the concrete XML documents. [20] has shown that evaluation of projection queries over probabilistic XML is possible in polynomial time (data complexity). This result by itself, however, does not imply PTIME algorithm for our setting, which is more complex: first, our model allows representation of *nested* DAG structures, rather than trees. For projection queries, this entails another level of complexity

[8] For clarity of presentation, we assume a total order on the expansions order. This assumption may be relaxed [14].

in finding matches serving as implementations, and projecting them over relevant nodes; second, an infinite number of flows may be represented by our probabilistic BP (serving as *schema*), due to possible recursive calls. Upon projection, the results set may be infinite as well, posing a further challenge.

In an XML setting, evaluation of projection queries with the *maximum* aggregation function may be done by a straightforward adaptation of selection queries evaluation algorithms. Thus, studies focused on the *sum* aggregation function. In contrast, we have shown that, in the settings of probabilistic BP, there are inherent difficulties in adaptation of algorithms for support of projection queries, even for *maximum* aggregation. The *sum* aggregation function is considered as future work.

Probabilistic Databases (PDBs) [11, 26] allow representation of uncertain relational information. They extend relational algebra to a probabilistic setting, and analyze the complexity of query evaluation in this setting. They show that finding the exact probabilities for query results is $\sharp P$-hard [11] (though top-k query evaluation is PTIME, by approximating results probabilities [24]). The hardness, however, stems from combination of projections (again using the *sum* aggregation function) with joins, absent in our context, and thus does not hold in our settings. In contrast, different difficulties rise in our setting, with the representation of the dynamic nature of flow and the possibly unbounded number of recursive (possibly dependent) activity invocations. In terms of the *possible worlds semantics* [11], the number of worlds in our model is infinite (rather than large, yet finite, in PDBs).

A variety of formalisms for probabilistic process *specifications* exist in the literature, such as Markov Chains [18], Probabilistic Recursive State Machines (PRSMs) [16], and Stochastic Context Free (Graph) Grammars (SCFG, SCFGG) [23]. To the best of our knowledge, projection queries over these models were not studied. As for selection queries, analysis of SCFGGs generally extends the theoretical analysis of Courcelle [10] on regular (non-stochastic) CFGGs, considering strongly expressive logics such as MSO (Monadic Second Order Logic). The complexity bounds obtained there are thus practically infeasible. For PRSMs, evaluation of strongly expressive query languages such as $PCTL^*$[25] is undecidable in general, and EXPTIME-hard [7] for some fragment. Other works consider linear-time formulas [21, 15], allowing to express only string-like properties. In contrast, our query language suggests a reasonable tradeoff between expressibility and feasibility [12].

Another advantage of using the query language studied here is its graphical and intuitive nature. Being similar to the graph-based view of processes used by commercial vendors for the specification of BPs [3], it is claimed to be more intuitive for BP developers than other query formalisms such as temporal logics and process algebras [2, 3].

7. CONCLUSION

We have studied in this paper, for the first time, top-k query evaluation for projection queries over probabilistic BP specifications. We have shown a PTIME (data complexity) algorithm for query evaluation in presence of independency between events dictating the BP flow, and studied the problem complexity in presence of dependency.

Furthermore, we have considered an application of our results to the analysis of BP execution traces, for recovering information missing from the traces.

We have limited the discussion in this paper to the *max* aggregation function, computing the rank of any given projection as the *maximal* likelihood of flows leading to it. Various other aggregation functions, and in particular the *sum* aggregation function, will be considered as future research.

8. REFERENCES

[1] S. Abiteboul and P. Senellart. Querying and updating probabilistic information in xml. In *Proc. of EDBT*, 2006.

[2] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *Proc. of VLDB*, 2006.

[3] C. Beeri, A. Eyal, T. Milo, and A. Pilberg. Monitoring business processes with queries. In *Proc. of VLDB*, 2007.

[4] S. Bhiri, W. Gaaloul, and C. Godart. Mining and improving composite web services recovery mechanisms. *Int. J. Web Service Res.*, 5(2), 2008.

[5] Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/.

[6] Oracle bpel process manager administrator's guide - configuring and viewing bpel process logs. http://download.oracle.com/docs/cd/E11036-01 /integrate.1013/b28982/logging.htm.

[7] T. Brazdil, A. Kucera, and O. Strazovsky. On the decidability of temporal properties of probabilistic pushdown automata. In *Proc. of STACS*, 2005.

[8] S. Cohen, B. Kimelfeld, and Y. Sagiv. Incorporating constraints in probabilistic xml. In *Proc. of PODS*, 2008.

[9] M. Consens and A. Mendelzon. The g+/graphlog visual query system. In *Proc. of SIGMOD*, 1990.

[10] B. Courcelle. The monadic second-order logic of graphs. *Inf. Comput.*, 85(1), 1990.

[11] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. of VLDB*, 2004.

[12] D. Deutch and T. Milo. Querying structural and behavioral properties of business processes. In *Proc. of DBPL*, 2007.

[13] D. Deutch and T. Milo. Type inference and type checking for queries on execution traces. In *Proc. of VLDB*, 2008.

[14] D. Deutch and T. Milo. Evaluating top-k queries over business processes. In *Proc. of ICDE*, 2009.

[15] K. Etessami and M. Yannakakis. Algorithmic verification of recursive probabilistic state machines. In *Proc. of TACAS*, 2005.

[16] N. Friedman, L. Getoor, D. Koller, and A.Pfeffer. Learning probabilistic relational models. In *Proc. of IJCAI*, 1999.

[17] W. Gaaloul and C. Godart. Mining workflow recovery

from event based logs. In *Business Process Management*, 2005.

[18] J. G. Kemeny and J. L. Snell. *Finite Markov Chains.* Springer, 1976.

[19] B. Kimelfeld, Y. Kosharovsky, and Y. Sagiv. Query efficiency in probabilistic xml models. In *Proc. of SIGMOD*, 2008.

[20] B. Kimelfeld and Y. Sagiv. Matching twigs in probabilistic xml. In *Proc. of VLDB*, 2007.

[21] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag, 1992.

[22] S. P. Meyn and R. L. Tweedie. *Markov Chains and Stochastic Stability.* Springer-Verlag, 1993.

[23] T. Oates, S. Doshi, and F. Huang. Estimating maximum likelihood parameters for stochastic context-free graph grammars. In *Proc. of ILP*, 2003.

[24] C. Re, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *Proc. of ICDE*, 2007.

[25] M. Reynolds. An axiomatization of pctl. *Inf. Comput.*, 201(1), 2005.

[26] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *Proc. of ICDE*, 2007.