Evaluating TOP-K Queries Over Business Processes

Daniel Deutch, Tova Milo
Tel Aviv University
{danielde,milo}@post.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 as they allow to optimize the BP, identify potential problems, and reduce operational costs. In particular, given a BP specification, identifying the top-k execution flows that are most likely to occur in practice out of those satisfying the query criteria, is crucial for various applications. To address this need, we introduce in this paper the notion of *likelihood* for BP execution flows, and study top-k query evaluation (finding the k most likely matches) for queries over BP specifications. We analyze the complexity of query evaluation in this context and present novel algorithms for computing top-k results to a query. We also experimentally demonstrate the efficiency of our algorithms. To our knowledge, this is the first paper that studies such top-k query evaluation for BP specifications.**

I. Introduction

A Business Process (BP for short) is a collection of logically related activities that, when combined in a flow, achieve a business goal. A BP usually operates in a cross-organization, distributed environment and the software implementing it is fairly complex. 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 [1]), 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.

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 "At which points of the flow is the user asked to relay her credit card details?", "In what ways may one reserve a travel package containing a flight and an hotel?", or "How is this done for travelers of a particular airline company, say British Airways?", etc.

A typical query engine is given as input the BP specification and an execution pattern of interest, and identifies, among the potential execution flows of the BP, all those having the structure specified by the pattern ([2], [3], [4]).

Note, however, that among all qualifying execution flows, some are typically more "interesting" than others. In particular, given a BP specification, identifying the top-k execution flows that are *most likely to occur in practice*, out of those satisfying the query criteria, 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, to personalize on-line advertisements, focusing on particular target audience, or to provide appealing package deals.

For instance, say that we obtain that a popular execution flow containing a British Airways reservation 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 result may strongly 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.

The importance of top-k query evaluation is enhanced by the fact that the total number of answers (qualifying executions flows) to a simple selection query may be extensively large, or even infinite when the BP contains recursion [2]. For instance, in the above travel agency example, if travelers may repeatedly search for appropriate flight/hotel, an unbounded number of times (each time changing their search criteria or choices), before finally making a reservation, then the number of answers to the above mentioned queries is also unbounded, as each distinct flow is a qualifying answer. Focusing only on the most likely traces may thus be crucial for analysis purpose.

To address the need for such top-k analysis of BP executions, we introduce in this paper the notion of *likelihood* for BP execution flows, and study top-k (most likely matches) query evaluation for queries over BP specifications. We analyze the complexity of query evaluation in this context and present efficient algorithms for top-k query computation. To the best of our knowledge, this is the first paper that studies such top-k query evaluation over BP specifications.

Our contribution here is threefold. First, we present a simple generic probabilistic model that allows to describe the possible execution flows of a given BP, and their likelihoods. Our model is a natural extension, to a probabilistic context, of the model of [2], [3], an abstraction of the BPEL standard (Business Process Execution Language [1]) 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 as-

signments/server states (and consequently guarding formulas/ truth values/ implementation choices) are more likely than others. Moreover, the likelihood of a particular choice (referred to as c–likelihood , for *choice likelihood*) may vary at different points of the run and may *depend* on the course of the run so far and on previously made choices. We exemplify scenarios that may not be captured by previously defined dependency notions (e.g. used in markov chains), but are expressible by our refined notion of c–likelihood . A likelihood of a full flow (called f–likelihood for *flow likelihood*) is composed out of c–likelihood values of choices occurring along the execution. *This probabilistic model is the first contribution of our work.*

Next, we consider queries. The query language that we consider selects execution flows of interest, using execution patterns. Execution patterns are an adaptation of the tree- and graph-patterns offered by existing query languages for XML and graph-shaped data, to BP nested DAGs. We study the problem of identifying, for a given BP and a query, the top-k execution flows with highest f–likelihood out of these satisfying the query. We distinguish several classes of c–likelihood functions, based on the level of inter-choices dependency they entail, and analyze the complexity of query evaluation for each of the classes. We show that for a large, common in practice, class of c–likelihood functions, the problem can be efficiently solved, and focus in particular on a practically common class of c–likelihood functions, namely functions that are of *bounded-memory*. Intuitively, with such function, the likelihood of a choice depends only on a bounded number of previously made choices. *The second contribution of this paper is a novel algorithm that computes the top-k execution flows conforming to a query, in presence of a bounded-memory c–likelihood function.*

Last, *the third contribution of this paper is an experimental study, indicating the efficiency of our evaluation algorithms.*

Note: The input for our analysis is (1) a BP specification and (2) a description of the c–likelihood function, representing the (conditional) probability of each implementation choice. The design/inference of both has been the focus of several previous works (see e.g. [5], [6]). We assume both are readily given, and their inference is outside the scope of this work.

Paper organization: We start in Section II, where we give an overview of related work. In Section III we formally define our model and the notion of top-k execution flows. In section IV we define several classes of likelihood function , and in sections V and VI present our algorithms for finding the top-k execution flows conforming to a query, for these classes. Section VII describes our implementation and experimental results. We conclude in section VIII.

II. Related Work

We give in this section a brief review of related work, highlighting the relative contributions of our results.

The query language and data model that we consider here are extensions, to a probabilistic setting, of those introduced in [2], [3]. They are argued there to be more intuitive for BP developers than other query formalisms such as temporal logics and process algebras, as they are based on the same graph-based view of processes used by commercial vendors for the specification of BPs [3]. The same arguments hold for our setting. A detailed comparison to related, non-probabilistic data models and query languages is given in [7], and we next review some of the related probabilistic models.

Probabilistic Databases (PDBs) [8], [9] and Probabilistic Relational Models (PRMs) [10] allow representation of uncertain information. However, they consider relational data and do not capture the dynamic nature of flow and the possibly unbounded number of recursive (possibly dependent) activity invocations. In terms of the *possible worlds semantics*, the number of worlds in our model is infinite (rather than large, yet finite, in PDBs). Extensions of PRMs to a dynamic setting, called dynamic PRMs, exist [11]. However, having here data of a particular restricted shape, common to execution flows, allows us to obtain practically efficient algorithms.

Probabilistic XML [12], [13] bears some resemblance to our model: the data is graph (tree) shaped, and it allows some dependencies between probabilistic events. However, our model is more complex: first, it represents nested DAG structures, rather than trees, entailing more intricate dependencies between events. Second, potentially infinite number of such nested DAGs are represented, due to possible recursive calls. In a sense, we consider here a probabilistic (recursive) *schema* rather than a given probabilistic document.

A variety of formalisms for probabilistic process *specifications* exist in the literature. Among these we mention Markov Chains [14], Probabilistic Recursive State Machines (PRSMs) [10], and Stochastic Context Free (Graph) Grammars (SCFG, SCFGG) [15]. PRSMs and SCFGGs describe nested structures similar to our model. However, they may not capture general probabilistic BPs, as they assume independencies (context freeness) between probabilistic events. Markov Chains (of order m) can express dependencies, but they model Finite State Machines and do not support the nested structure (and consequently the more intricate dependencies) considered here.

Work on *querying* SCFGGs generally extends the theoretical analysis of Courcelle [16] on regular (non-stochastic) CFGGs, and thus uses strongly expressive logics such as MSO (Monadic Second Order Logic). As a consequence, the complexity bounds obtained there are completely infeasible. As for PRSMs, most works address only linear-time formulas (LTL, ω-regular expressions [17]). This essentially means that only string-like properties [18] of the process may be queried. As for stronger formalisms such as $PCTL^*$[19], its evaluation over PRSMs was shown to be EXPTIME-hard [20] for some fragment, and undecidable for the entire language. On the one hand, the query language we use here allows representation of interesting properties inexpressible in LTL [7], and on the other hand allow for efficient evaluation. Moreover, we consider *top-k* query evaluation, that to our knowledge has not been addressed by any of these previous works.

Complementary to our work are works that query and analyze run-time *logs* of BPs [21], [22], [23]. In contrast, our work allows for *static analysis*, working on the *BP specification*.

III. PRELIMINARIES

We present in this section the formal definitions for our model and query langauge, accompanied by examples.

BP specification*:* As a running example, we consider here a web-based travel agency. The web-site users first choose between several usage options, such as searching for a trip, viewing reviews, or contacting the web-sites owners. If they choose to search for a trip, they can then 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.

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 $start$, $chooseTravel$ 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. For instance, at the $start$ activity, the user may choose between three possible ways of usage. By setting $\$usage$ to be "search travel", a $chooseTravel$ activity isinvoked; it has three possible implementations $F3$, $F4$, $F5$, 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 $F3$ 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 $F2$. Other options here are confirmation and cancellation.

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. 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.

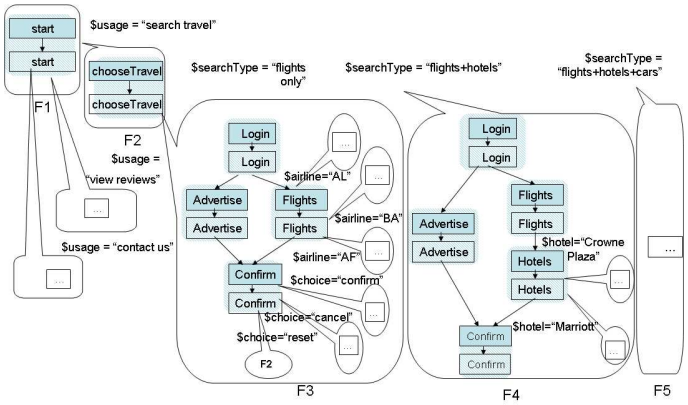

Fig. 1. Business Process

Definition 3.1: 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 of the 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 that 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 3.2: 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 $\{F1, F2, F3, F4, F5\}$, $s_0 = F1$ 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 the chosen implementation for the compound activities. Fig. 2(a) depicts an example execution flow of the travel agency process (ignore, for now, the numbers annotating the different choices). Zoom-in edges (denoted in the figure by dashed arrows) connect the activation and completion nodes of compound activities to the start and end (resp.) nodes of the chosen implementation.

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

Definition 3.3: Given a BP specification $s = (S, s_0, \tau)$, g is an execution flow (EX-flow) of s if:

- g consists only of the specification's root activity node s_0, or,
- if g' is an execution flow of s, and $g' \rightarrow g$. That is, 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.

g is called a *full flow* if it cannot be further expanded (that is, it contains the internal flow for each compound activity node in it), and a *partial* flow otherwise. The set of all full *EX-flows* defined by a BP s is denoted $flows(S)$.

Note: At run time, a given EX-flow may be obtained via different expansion sequences that vary in the order in which *parallel* activities are expanded. To simplify the presentation we will assume below some *total order* on the activities of each EX-flow, consistent with their ancestor relationship, and assume that activities are expanded following this order. (We stress that this is only for presentation considerations - our results extend to the general context where multiple expansion orders are possible, and are discussed below). In this case, for every partial EX-flow e, the activity that is next to be expanded in e is unique and well-defined, and the execution course is determined by choices of implementations.

Likelihood*:* The set of possible execution flows of any given specification may be extensively large, and even infinite in case of recursive specifications. However, some execution flows are more likely than others to occur in practice. We capture the notion of execution flow likelihood through two kinds of likelihood functions. The first, denoted c–likelihood (choice likelihood) function, associates a likelihood with each implementation choice (guarding formula) *given a description of the flow preceding the point where the corresponding implementation choice took place*. The value assigned by the function represents the probability that the formula holds, given the flow history. To model this, we assume the existence of a c–likelihood function δ that, given a partial EX-flow e which describes the course of the run so far, and a formula f that guards an implementation of the activity that is to be expanded next, returns some number $0 \leq \delta(e, f) \leq 1$. $\delta(e, f)$ is the likelihood that f will hold at the point of run specified by e. As we have assumed above that at run-time, exactly one implementation is chosen for each activity instance, we require that the sum of c–likelihood values over all guarding formulas of a single activity name is 1. For now, we assume that δ is given as an oracle; its concrete representation is discussed separately below.

Example 3.1: Consider for example the execution flows of our travel agency BP, depicted in Figure 2. Figure 2(a) depicts an execution flow containing a "searchTravel" usage whose c–likelihood is 0.9, followed by a choice of a "flights+hotels" search (whose c–likelihood is 0.6). The execution flow of Figure 2(b), on the other hand, contains a choice of a "flights only" search (whose c–likelihood is 0.3). Then, both flows contain choices of the "British Airways" ("BA") airline. However, for the flow in Fig. 2(a), the c–likelihood of the "BA" choice is 0.4, where in (b) its c–likelihood is 0.7. It exemplifies that the $c - likelihoods$ of different choices may be *dependent* on the flow preceding it. Additionally, note that the execution flow of (b) contains a second instance of $chooseTravel$, obtained as a consequence of a "reset" choice. The $c - likelihoods$ of the different choices for this instance of $chooseTravel$ are different from the c–likelihood of the choices for its first instance (0.7 for "flights+hotels", compared with 0.6 at the first instance), again exemplifying a dependency on flow history, this time between multiple instances of the same activity name.

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 set of formulas guarding implementations that were chosen during the execution; the joint likelihood of these formulas thus stands as the execution flow likelihood. Formally,

Definition 3.4: Given a BP s rooted at s_0 and a c–likelihood function for it, the *f–likelihood* 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, f–likelihood $(e) = 1$.
2) Else, if $e' \rightarrow e$ for some EX-flow e' of s, then f–likelihood (e) = f–likelihood (e') $\times$ c–likelihood (e', f), where f is the formula guarding the implementation that is added to e' to form e.

Example 3.2: To continue with our running example, let us consider the importance of the flow depicted in Figure 2(a). The f–likelihood of a partial flow e''' containing only the direct internal flow of the root $start$ activity (searching for a trip) is 0.9. Given such history, the c–likelihood of a search for flights & hotels is 0.6, so we obtain $f\text{-}likelihood(e'') = f\text{-}likelihood(e') \times 0.6 = 0.9 \times 0.6 = 0.54$, where e'' contains the choice of $flights + hotels$ choice. Given such search, the c–likelihood of the British Airways choice is 0.4. Thus $f - likelihood(e') = f - likelihood(e'') \times 0.4 = 0.216$, where e' is the partial EX-flow containing the "BA" selection as well. Then, with the choice e.g. of $\$hotel = Marriott$ and $\$choice = confirm$, we get $f - likelihood(e) = f - likelihood(e') \times 0.7 \times 0.5 = 0.216 \times 0.7 \times 0.5 = 0.0756$.

Queries*:* As mentioned in the introduction, the query language that we use was originally introduced in [2] and its semantics is extended here to support top-k queries. Queries are defined using *execution patterns*, which are adaptations of the tree- and graph-patterns, offered by existing query languages for XML and graph-shaped data ([24]), to nested DAGs. Edges in a pattern are either regular, interpreted over

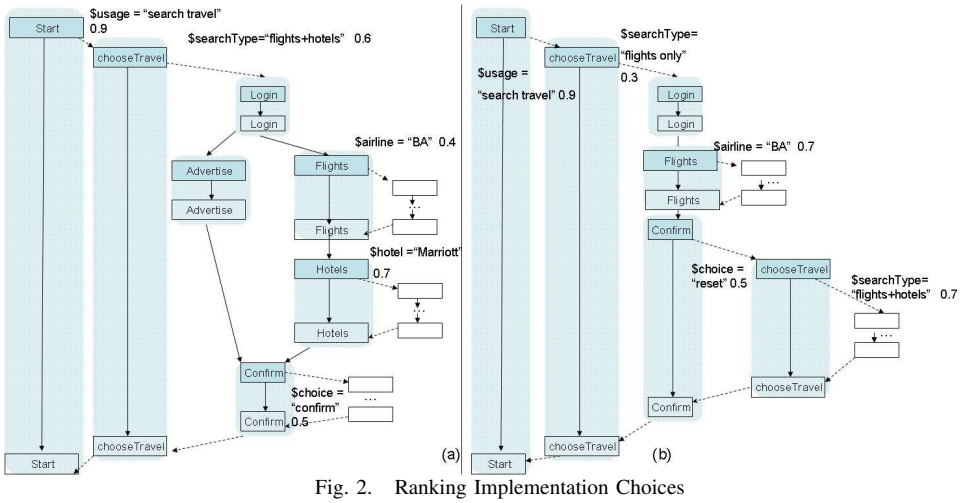

Fig. 2. Ranking Implementation Choices

edges, or transitive, interpreted over paths. Similarly, activity pairs may be regular or transitive for searching only in their direct internal flow or zooming-in transitively inside it, respectively. Moreover, activity nodes may be marked by a distinct "Any" symbol, allowing match against execution flows nodes bearing any label.

Definition 3.5: 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 $\mathcal{A} \cup \{\text{ANY}\}$. Nodes labeled by compound activity names may be additionally annotated by a guarding formula. T is a distinguished set of activity pairs and edges in $\hat{e}$, called *transitive* activities and edges, resp.

Consider, for example, the query in Figure 3, that selects EX-flows where the user reserved a British Airways flight. In other words, the query seeks for EX-flows containing a $start$ activity, in which (possibly indirect) implementation, $chooseTravel$ appears, and within its implementation, a choice of "BA" as airline is made and then confirmed. Syntactically, note first the double bounding of the $start$ activity-pair, denoting *transitive*, unbounded, zoom-in into its internal execution flow, searching for a $chooseTravel$ activity at any nesting depth (corresponding to any number of searches). The doubly-lined edges connected to the $start$ activity are *transitive edges*, and may be matched against any flow of activities. (Indirectly) following the $chooseTravel$ activity, the pattern requires a flights search, in which a British Airways choice is made. The choice is required to be confirmed, possibly after further choices of hotels and/or cars. This sequence of choices may match the transitive edge between the $hotels$ and $confirm$ activities.

To evaluate a query, the EX-pattern is matched against a given EX-flow. Such match is represented by an *embedding*.

Definition 3.6: 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; however, a node labeled by any can be mapped to nodes with any activity name. If a compound activity node in p is annotated by a guarding formula f then the implementation of the matching node in e must be guarded by f.
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 flow 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 $[n, m]$.

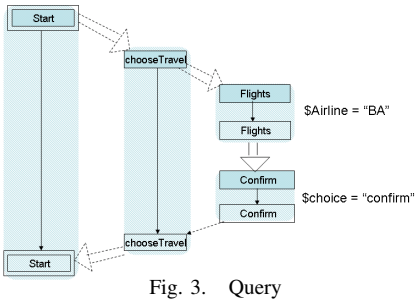

Fig. 3. Query

Query results*:* When evaluated over a BP specification s, a query q restricts the attention to EX-flows that *comply to the pattern*, i.e. to EX-flows e s.t. there exists some embedding of q into e. For a BP s and a query (EX-pattern) q, we use $q(s)$ to denote the subset of the possible EX-flows of s that comply to the pattern. As mentioned above, the set $q(s)$ of results may be very large, or even infinite. For instance, in our example, any flow containing any number of resets, followed by a confirmed "BA" choice, qualifies. To focus on the more common flows, we use $top-k(q, s)$ to denote the set of k EX-flows in $q(s)$ having highest f–likelihood .

Technical Remark: We have required above that no two guarding formulas of the same activity may be satisfied concurrently, and consequently that the sum of likelihoods of guarding formulas for each activity name is 1. While these assumptions comply with an intuitive concept of BP specification, they are not necessary for our results. Indeed, we use below, as a tool, a model for BP without such assumptions.

IV. CLASSES OF C-LIKELIHOOD FUNCTIONS

Recall that the likelihood of execution flows, f–likelihood , is dictated by the c–likelihood function used for the different implementation choices. As a consequence, the properties of a c–likelihood function greatly affects the complexity of finding the top-k most likely flows. A likelihood function receives as input both a guarding formula and a partial flow representing the "history", i.e. flow thus far. Intuitively, a c–likelihood function is simple if it's "local", i.e. it is in fact a function of the guarding formula solely, or a function of the formula along with only a small amount of history that affect its value.

To capture that, we define a notion of a c–likelihood function *memory*, and corresponding three classes of functions with varying memory properties.

A. Memory-less

The simplest class of c–likelihood functions is the *memory-less* class. δ is *memory-less* if for each formula f and for each two partial flows (histories) e, e', $\delta(f, e) = \delta(f, e')$. This means that the c–likelihood function is indifferent to history, and that the choices are all independent.

Example 4.1: Consider, for example, the memory-less c–likelihood function depicted in Table I. For each entry, the "P" column stands for the choice's c–likelihood . Each entry corresponds to a variable value, hence to a choice of a guarding formula / implementation. The c–likelihood of each choice is independent of other choices made during the execution.

$\$usage$	$P(\$usage)$		
"search travel"	0.9		
"view reviews"	0.05		
"contact us"	0.05		
$\$searchType$	$P(\$searchType)$	$\$airline$	$P(\$airline)$
"flight only"	0.5	"BA"	0.7
"flight+hotel"	0.25	"AF"	0.2
"flight+hotel+car"	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 I

MEMORY-LESS C–LIKELIHOOD FUNCTION

B. Bounded memory

A more general and more realistic class of c–likelihood functions captures the common case where the c–likelihood of any given choice may depend on the execution history, but only in a bounded manner.

One common way (e.g. in Markov chains of order m [14]) of defining bounded memory distributions uses a "sliding window" of size m, which in our setting would translate into dependency of each choice on the m last choices made prior to it. However, this approach may lead to omitting some important choices from the memory. Consider, for instance, a recursive BP, and a choice that preceded the recursive loop. For executions in which the user re-entered the loop for more than m times, a function that uses a sliding window of constant size m will ignore any choice that had occurred prior to the recursive loop.

To overcome this, we define a richer notion of memory, by using a set of "sliding windows", one for each activity name. That is, there exists some bound m such that, at most, the m last choices (if at all) made *for each activity* actually affect the c–likelihood of f. I.e., whether f holds or not is *independent* of further previous choices, given this history.

Re-considering our travel agency BP, assume that the choice of hotel is dependent on the last (previous) choice of search type, and on the last choice of airline; the memory bound for the hotels activity, in this case, is 1. In contrast, if the choice of hotel depended on the two previous choices of searchType, then the memory bound would be 2. The notion of memory relates to the m last occurrences of *each activity name*.

More formally, given a BP s, we say that δ is *bounded-memory* if there exists some finite bound m, s.t. 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* m previous expansion steps, $\delta(e, f) = \delta(e', f)$.

Note that m is only a bound, and some choices may be completely (conditionally) independent of each other, as exemplified below.

Example 4.2: An example for a bounded-memory c–likelihood function over the implementation choices in our travel agency BP is given in Table II, where the c–likelihood of each choice, given its relevant history, is given. $P(\$x \mid \$y)$ stands for the c–likelihood of (values of) $\$x$ given a flow history containing specific values of $\$y$. Note that a choice of a variable value (hence an implementation choice) is also implicitly dependent on other choices that are necessary for its relevance: for example, although not put explicitly in the table, the choice of a hotel depends on $\$searchType$ being either "flights+hotels" or "flights+hotels+cars" (otherwise no hotels can be chosen at all). This information appears in the application description (Fig. 1) and is redundant in the table.

An example for conditional independency is captured by the likelihood of the confirmation choices, that depends only on the choice of airline, and given such choice, is independent of the choice of hotel. This exemplifies that the memory-bound is not necessarily tight, and some choices may be independent of each other. Note as well that due to the specification recursive nature, the truth values of some guarding formulas occurrences may depend not only on values of other formulas, but on previous values assigned to the same formula. In our example, the likelihood of $\$searchType$ being any specific type depends on the user's previous choice for a $\$searchType$ (the previous choice is denoted by $\$searchType(1)$, and the current choice by $\$searchType(2)$, $\perp$ indicates that there was no previous choice for a search type in this session, "f", "f+h","f+h+c" stand for "flights", "flights+hotels", "flights+hotels+cars" resp.).

C. Unbounded-memory

In general, there exist c–likelihood functions that do not fall under the bounded-memory. We refer to these as *unbounded-memory* ranking functions.

Consider for example a c–likelihood function where the c–likelihood of the choice of a given hotel depends on exactly how many times the "reset" option was previously chosen. Here, an unbounded number of choices (for the same activity name) must be remembered for the computation of the ranking function. In practice, such scenarios are rare (see e.g. [25]).

$\$usage$	$P(\$usage)$
"search travel"	0.9
"view reviews"	0.05
"contact us"	0.05

$P(\$searchType(2) \mid \$searchType(1))$	$\perp$	"f"	"f+h"	"f+h+c"
"f"	0.3	0.1	0.7	0.7
"f+h"	0.6	0.7	0.3	0.1
"f+h+c"	0.1	0.2	0	0.2

$P(\$Airline \mid \$searchType)$	"flights only"	"flights +hotels"	"flights +hotels+cars"
"BA"	0.7	0.4	0.6
"AF"	0.1	0.4	0.3
"AL"	0.2	0.2	0.1

$P(\$Hotel \mid \$Airline)$	"BA"	"AF"	"AL"
"Marriott"	0.7	0.6	0.1
"HolidayInn"	0.05	0.1	0.7
"CrownePlaza"	0.25	0.3	0.2

$P(\$choice \mid \$Airline$	"BA"	"AF"	"AL"
"reset"	0.5	0.4	0.8
"confirm"	0.5	0.6	0.2

TABLE II

BOUNDED-MEMORY RANKING

D. Summary of Our Results

We next summarize the main results of this paper. The formal theorems/algorithms are given in the following sections.

Memory-less*:* For memory-less c–likelihood functions, we provide a PTIME (data complexity) algorithm for query evaluation , and show that no PTIME (combined complexity) algorithm exists, unless $P = NP$. (Theorems 6.2, 6.4).

Bounded-Memory*:* For bounded-memory c–likelihood functions, we show that no PTIME data complexity algorithm exists, unless $P = NP$, and give an EXPTIME algorithm (combined complexity). (See Theorem 6.5.) Nevertheless, we provide strong heuristics that lead to a feasible execution time, even for large-scaled specifications.

Unbounded-Memory*:* We show that the results may not be extended to unbounded-memory c–likelihood functions, as finding the top-k flows conforming to a query is, in general, undecidable (See Theorem 6.7)

V. QUERY EVALUATION

We now turn to describing in details our results.

A. General Framework

Given a BP s, a c–likelihood function δ, and a query q, the top-k query results are computed in two steps.

1) First, we construct a BP s' and a c–likelihood function δ' such that s' *captures* the set of query results $q(s)$. Formally, s' is defined such that (i) $flows(s')$ is the same, up to some renaming function ρ over its activity names, as $q(s)$, and (ii) for all $e \in flows(s')$, the f–likelihood of e (w.r.t. δ') equals to the f–likelihood of $\rho(e)$ (w.r.t δ). Furthermore, the algorithm is generic: if δ is memory-less (memory-bounded), so is δ'.
2) Next, depending on the type of δ (memory-less, bounded-memory), we generate from s' a new specification s'' whose set of flows corresponds exactly to the top-k query results. Finally, the renaming function ρ may be applied over flows of s'' to obtain the i'th ranked result for any $i \leq k$.

In the remainder of this section we present an algorithm that performs step 1. Step 2 is described in the next section.

B. Step 1 - Computing all query results

The algorithm is a natural extension of the BP query evaluation algorithm presented in [4], which did not consider the likelihood of execution flows, to take probabilities into consideration. We briefly recall below the algorithm of [4], highlighting the adaptation to a probabilistic setting.

Intuitively, the algorithm computes a BP s' that is an "intersection" of the BP s and the query q. To see how this works, let us consider first queries (EX-patterns) without transitive nodes. Our algorithm uses a function EMBED, 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 activity names to g; the activity name labeling each of its nodes encodes the identifier and label of the original node in g, along with the labels of all nodes of p mapped to it (if any).

The algorithm begins by embedding the pattern root r' in the specification root r, creating a new root $[r, r']$. Then, we use EMBED to embed the implementation of r', in all direct implementations of r. Each embedding returned by EMBED will be an implementation of the $[r, r']$, and it will be guarded by its origin's guarding formula. The c–likelihood function δ' assigns to each guarding formula the same value assigned to it by δ. For each matched compound activity of the pattern, we then match recursively its implementation (in the pattern), to any implementation (in the BP) of any 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. As a final step we perform a "garbage collection", removing each activity for which all implementations lead to failure (then activities that only lead to these activities, etc.).

If the query 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, and recursively match them to the BP implementations. Note that since patterns may have several splits, an implementation of a compound activity may be matched to different sub-patterns and thus appear in the result multiple times. Consequently, the same guarding formula will guard the multiple occurrences, possibly yielding sum of likelihoods > 1. As noted at the end of Sect. III, our results all apply to this extended notion of BP.

Example 5.1: A BP s' capturing the result of the query of Figure 3 when applied on the BP of Figure 1, is given in Figure 4 (without the c–likelihood). Note that s' enforces a choice of "British Airways" as the user's final choice - the only way to confirm is through the $confirm''$ activity, in $F4$ ($F6, F8$), and if such implementation is chosen, then the $Flights''$ activity within it allows only the choice of "British Airways". Before that, the user may choose any of the other implementations, but without confirming ($confirm'$ allows only reset-

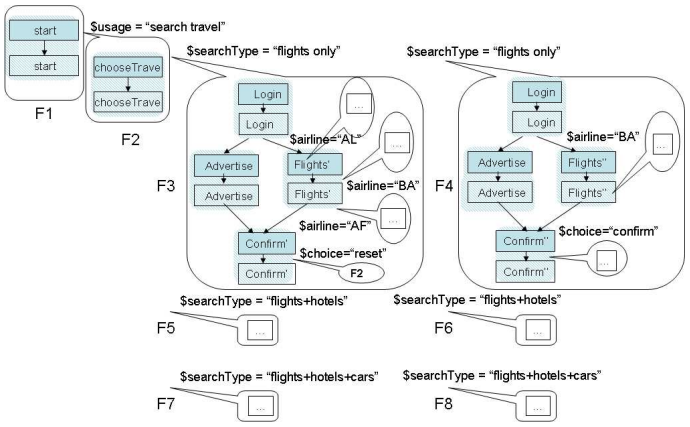

Fig. 4. All Matches

ting). $confirm', confirm''(Flights', Flights'')$ are renamed as $confirm$ $(Flights)$.

Complexity*:* The time complexity of the matching algorithm is polynomial in the size of the specification, with the exponent determined by the size of the query. The output is a new BP specification s', representing all matching flows, and its size is, again, polynomial in the size of the specification, with the exponent determined by the size of the query.

VI. Finding top-k EX-flows

We next consider step 2, namely the computation of the top-k flows of the BP s' that captures $q(s)$. Note that while step 1 is generic and applies uniformly to all classes of c–likelihood functions, in step 2 we consider the classes separately. We start by considering memory-less c–likelihood functions, then extend our discussion to other classes.

A. Memory-less c–likelihood functions

We present below an efficient algorithm for computing a compact representation of the top-k qualifying flows, for memory-less c–likelihood functions. It exploits the "local" nature of memory-less functions, where the c–likelihood of each formula (implementation choice) is independent of the choice of other formulas. The algorithm is a form of Dynamic Programming, where a table containing the sub-flows with highest f–likelihood originating from each compound activity is gradually computed. Recall that we have defined above (Definition 3.4) f–likelihood only for EX-flows that originate from the BP *root* activity. However, the definition naturally extends to sub-flows originating from an arbitrary compound activity a (treating a as a root).

Our algorithm is based on the following Lemma.

Lemma 6.1: For every memory-less c–likelihood function δ, a BP s and a compound activity a in s,

1) There exists a best ranked (top-1) EX-flow originating at a that does not contain another occurrence of a.
2) There exists a $j+1$'th ranked EX-flow originating at a such that for any occurrence of a in it, the sub-flow rooted at the latter is one of the top-j EX-flows of a.

Proof: The proof for the top-1 EX-flow follows from the observation that for each flow that contains an occurrence of a we may, without reducing f–likelihood , eliminate the cycle between the first and the last expansions of a, replacing the internal flow of the first a by that of the last one. Similarly, for a $j+1$'th EX-flow that violates item 2 above we may replace the sub-flow rooted at the latter a by one of the $top-j$ EX-flows of a. ■

The above lemma implies that when searching for the top-k EX-flows, it suffices to examine a *finite number* of EX-flows. Thus, a simple algorithm for finding the top-k EX-flows is to enumerate all these EX-flows, compute their f–likelihood , and pick out the top-k ones. Note however that the number of EX-flows examined by this naive algorithm may be *exponential* in the size of the BP s. To avoid examining all of them we propose below an optimized PTIME algorithm that focuses only on the essential flows.

The `TOP-K` algorithm*:* Given a BP $s' = (S', s'_0, \tau')$ with a c–likelihood function δ' (obtained in the previous section), our algorithm builds a new BP $s'' = (S'', s''_0, \tau'')$ (with δ'') whose EX-flows capture the top-k EX-flows of s (and their likelihoods). More precisely, for each compound activity $a \in s'$, s'' has k corresponding compound activities $[a,i]$, for $i = 1 \ldots k$. Each such activity $[a,i]$ in s'' will have a single EX-flows originating from it, denoted $flow([a,i])$, which *captures* the i^{th} ranked EX-flow originating from a in s', in the sense that (1) the latter is obtainable from $flow([a,i])$ by simply removing the number annotations of the activity names, and (2) the two EX-flows have precisely the same f–likelihood (in s' and s'' resp.).

Note that once such a BP s'' is constructed, the top-k flows (and their f–likelihood) can be effectively enumerated, in time linear in the flow size, by simply expanding its root activity. We next show that s'' can be computed efficiently.

The `TOP-K` algorithm that constructs s'' is depicted in Figure 5. Initially s'' contains only the root activity with no implementation (line 1). To build s'', the algorithm uses an array, called $rank$, with an entry $rank[a,i]$ for each activity $a \in s'$ and $i = 1 \ldots k$. $rank[a,i]$ records the f–likelihood of the flow originating at $[a,i]$ in s'' (which is also the f–likelihood of the i^{th} ranked flow originating in a in s'.) `TOP-K` uses the subroutine `FLOW` (line 7) that, given an activity $[a,i]$, determines the implementation τ'' of $[a,i]$, as well as its entry at the $rank$ array. Finally, the implementation of the root activity s''_0 is set (line 8) to include the computed implementations of the $[s'_0, i]$, $i = 1 \ldots, k$, activities, that represent the top-k flows.

To decide the implementation τ'' of $[a,i]$, `FLOW` uses (in line 2) a subroutine `CANDIDATES` (to be explained below), which tells which possible implementations d (with guarding formulas f) may be relevant to $[a,i]$. For each of these possible (formula,implementation) pairs, the value of $rank[a,i]$ (i.e. f–likelihood of the flow originating at $[a,i]$ in s''), if this implementation would be chosen, is computed, and the one yielding maximal value is chosen (lines 8-13). The implied rank is the multiplication (line 8) of the rank of the compound activities appearing in the implementation (computed recursively using `FLOW`, in lines 5-6) and the c–likelihood of

	TOP-K[input:$(S', s'_0, \tau'), \delta', k$ output:$(S'', s''_0, \tau''), \delta''$]
1	$s''_0 = s'_0$; $S'' = \{s''_0\}$; $\tau''(s''_0) = \emptyset$;
2	**for each** activity $a \in S'$ and number $i = 1 \dots k$
3	$rank([a, i]) = 0$;
4	**for each** atomic activity $a \in S'$
5	$rank([a, 1]) = 1$;
6	**for** $i = 1 \dots k$
7	call $FLOW(s_0, i)$;
8	$\tau''(s_0) = \tau''(s_0) \cup \tau''([s_0, i])$;
9	**end for**

	FLOW[input:a, i]
1	**if** $\tau''([a, i])$ is not defined already
2	Mark $\tau''([a, i])$ as defined;
3	$candidates = CANDIDATES(a, i)$;
4	**if** $candidates \neq \emptyset$
5	**for each** activity $[b, j]$ appearing in $candidates$,
6	call $FLOW(b, j)$;
7	**for each** $(f, d) \in candidates$,
8	let $cRank(f, d) = \delta(f) \times \Pi_{[b,j] \in d} rank[b, j]$;
9	let (f_{max}, d_{max}) be the pair with maximal $cRank$;
10	$rank[a, i] = cRank(f_{max}, d_{max})$;
11	**if** $rank[a, i] > 0$
12	$\tau'([a, i]) = \{(f_{max}, d_{max})\}$;
13	$S = S \cup \{d_{max}\}$;
14	$\delta''(f_{max}) = \delta'(f_{max})\}$;
15	**end if**; **end if**; **end if**

	CANDIDATES[input:a, i output:$candidates$]
1	$used = \{(f, d) \mid (f, d) \in \tau'([a, j]), j < i\}$;
2	$candidates =$
	$\{(f, d') \mid (f, d) \in used$ and d' is obtained from d by increasing the index of one activity in d. $\}$
	$\cup \{(f, d') \mid (f, d) \in \tau(a)$ and d' is obtained from d by attaching the index 1 to all activities. $\}$
	$- used$;
3	return $candidates$;

Fig. 5. The TOP-K Algorithm

the formula guarding this implementation.

The subroutine CANDIDATES determines the potential (formula,implementation) pairs based on Lemma 6.1: the only implementations that need to be considered for the i^{th} ranked flow originating in an activity a are those that hadn't been used yet for the lower ranked flows of that activity, or are "worse" (f–likelihood -wise) than them in a minimal way (line 2).

To summertime,

Theorem 6.2: Given a BP s, a memory-less ranking function δ, and a query q, one can compute a compact representation of $top{-}k(q, s)$, in polynomial time (data complexity).

Proof: The PTIME algorithm starts by applying the algorithm of step 1 to obtain $s' = q(s)$, then apply the TOP-K algorithm over the obtained specification s', to obtain a compact representation s'' of only the top-k matches. The PTIME complexity of step 1 was shown above. The PTIME complexity of TOP-K is due to the fact that each $[a, i]$ pair is processed by FLOW at most once (line 1), and at most $k \times |S|$ candidate implementations are considered for it, yielding a complexity of $O(k \times |S|^2)$. ■

Example 6.3: To illustrate the algorithm, let us consider the BP $s' = q(s)$ of Figure 4, assuming the memory-less c–likelihood function given in Table I above. Consider

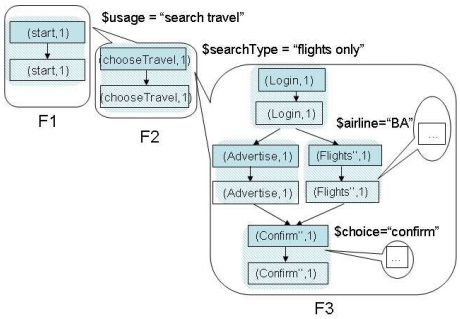

Fig. 6. TOP-1 BP for the memory-less ranking

the execution of TOP-K for $k = 1$. For each graph F_i, $i = 1 \dots 8$, of the BP, we use below F_i^1 to denote the graph obtained from F_i by replacing each activity name a by $(a, 1)$. TOP-K starts by looking for the most likely flow rooted at $start$. $(\$usage = "searchtravel", F_2^1)$ is the first candidate to be examined. We compute the rank of F_2^1, and its obtained rank will be multiplied by the c–likelihood of $\$usage = "searchtravel"$ which is 0.9. Next, we look at the only activity of F_2^1, $searchTrip$. Its candidates set contains $(\$searchType = "flight\ only", F_3^1(F_4^1))$, $(\$searchType = "flight\ +\ hotel", F_5^1(F_6^1))$,$(\$searchType = "flight\ +\ hotel\ +\ car", F_7^1(F_8^1))$. Next, the top-1 flow rooted at any activity of F_3^1 is computed, and its f–likelihood will be multiplied by $\delta(\$searchType = "flight\ only") = 0.5$. This is computed by the recursive calls to FLOW in lines 5-6. For $Flights'$, the top-1 flow is the one corresponding to $\$airline = "BA"$ (whose c–likelihood is 0.7). For $Confirm'$, the only option is to reset; that causes a multiplication by the $cRank$ of the single activity of F_2^1, $(chooseTravel, 1)$, which was set to 0 in line 3 of TOP-K. Thus, the computed rank for F_3^1 is 0. In contrast, when we compute the rank for F_4^1 (again, its rank will be multiplied by the c–likelihood of its guarding formula, $\delta(\$searchType = "flight\ only") = 0.5$), we obtain a single option for an airline (for implementation of $Flights''$), and for confirmation, obtaining a likelihood of $0.7 \times 0.4 = 0.28$. Thus, the overall $cRank$ (line 10) of the second candidate $(\$searchType = "flight\ only", F4^1)$ is $0.5 \times 0.7 \times 0.4 = 0.14$. Similarly, the $cRank$ of the other candidates is computed, and for this example, none exceeds 0.14. Thus, $f{-}likelihood(start, 1) = 0.9 \times 0.14 = 0.126$. Note that the option of reset (through F_3^1) will still be considered for computation of second (and on)-ranked traces, now multiplying by $cRank(chooseTravel, 1) = 0.14$.

Fig. 6 depicts the BP obtained for TOP-1 query evaluation.

To conclude we also show that, unless $P = NP$, no algorithm whose time complexity is also polynomial in the query size exists. First, we define the decision problem of BEST-MATCH, that given a BP specification, a c–likelihood function, a query, and a bound B, tests for the existence of a match whose f–likelihood is greater than B.

Theorem 6.4: BEST-MATCH is NP-complete w.r.t. the query size, even for memory-less c–likelihood functions.

The hardness proof works by reduction from the problem of testing if the answer of $q(s)$ is empty, shown in [7] to be NP-hard. The NP algorithm uses Lemma 6.1 to guess a solution

of bounded size. We omit the details for space constraints.

B. Bounded-memory c-likelihood functions

Bounded-memory c–likelihood functions pose further challenges, as the c–likelihood of each choice may depend on a number of other choices. As a consequence, the computation above needs to be refined. We start by presenting a simple variant of the algorithm, then consider possible optimizations.

The `BOUNDED-TOP-K` **algorithm***:* The general idea of the algorithm is to create, given the BP s' obtained in step 1, and δ' with a memory bounded by m, a new BP s'', with a new, memory-less, c–likelihood function δ'', such that s' and s'' have essentially the same set of flows with the same f–likelihood . As the new c–likelihood function δ'' is memory-less we can now apply the algorithm `TOP-K` described above to find the top-k flows in s''. To create this memory-less function, we annotate the activity names in s', "factoring" within the names all information required for the computation of c–likelihood of formulas, namely a pre-condition vector, including the m last choices for all activities. Additionally, the new activities names also contain post-condition vectors, necessary to assure consistencies between pre-conditions assumed by activities and what has happened in their predecessors.

Given BP s' with activities $a_1 \dots a_n$, and assuming a memory bound of m over δ', the activities names in s'' are tuples of the form $(a, pre = [pre_1^1 \dots pre_n^1, \dots, pre_1^m \dots pre_n^m], post = [post_1^1 \dots post_n^1, \dots, post_1^m \dots post_n^m])$ where a is an activity name, pre_j^i denotes a formula guarding the implementation chosen for a_i in its previous $(j)^{th}$ expansion, prior to expanding a, and $post_j^i$ denotes a formula guaranteed to guard the implementation chosen for a_i in its next j^{th} expansion. The idea is that pre encodes all information required for computing c–likelihood of a's guarding formulas, and $post$ encodes all information required for activities that follow a in the flow. We use $pre_j^i = \perp$ if a_i was not expanded j steps before the flow reaches a, and $post_j^i = \perp$ if a_i will not be expanded j steps before the execution of a is done.

Next we construct the implementations of $(a, pre, post)$ in s'. For each implementation F_i of a in s (guarded by f_i), we create a set of new implementations. Each implementation is obtained by annotating each activity b in F_i with vectors of pre and post conditions. The requirements on annotations of nodes are as follows: (a) if r is the root of an implementation F_i of $(a, pre_a, post_a)$ guarded by f_i (note that many such activity names are created for any activity name a, differing in their pre- and post-condition vectors), then the pre-condition of r is obtained from pre_a, shifted by one step, recording F_i (and possibly deleting some formula from memory, if reached the bound), (b) if there exists an edge from some node n to some node n' in F_i, the post-condition annotating n complies with the pre-condition annotating n', and (c) if e is the end node of F_i, the post-condition annotating it complies with $post_a$.

Complexity*:* The algorithm presented above is polynomial in k and exponential in $\mid s' \mid \times m$ (with m being the memory bound). Unfortunately, no polynomial algorithm is likely to exist in this setting, as the following theorem holds.

Theorem 6.5: Given a BP s with a bounded-memory c–likelihood function δ and a query q BEST-MATCH is NP-complete w.r.t. the size of s.

The hardness proof here works by reduction from probability computation over bayesian networks, known to be NP-hard. The NP algorithm again guesses a solution of bounded size, based on Lemma 6.1. (Details omitted.)

However, the pathological scenarios that lead to this NP-hardness are not necessarily typical. We next optimize our EXPTIME algorithm, by identifying common cases where more efficient processing is possible.

Optimization*:* The exponent of the `BOUNDED-TOP-K` algorithm is determined by the sizes of the "memory" vectors. Thus, we aim at reducing their size. In practice, the required memory varies between activities and may be much lower than the sum of memory bounds over all activities, used in the algorithm above, due to (conditional) independencies that hold for at least some of the formulas. Each activity has in fact an *actual memory*, which is the amount of memory required so that its guarding formulas will be independent of all other formulas, given its memory. The goal of our optimizations is thus to bring the amount of memory kept for each activity to be as close as possible to its actual required memory.

First, observe that a formula f can be omitted from the $post$ vector of $(a, pre, post)$ if it does not appear in the pre vector of any activity name that is reachable (through zoom-in/flow edges or a combination thereof) from a. Thus, reducing the sizes of the pre vectors will also allow reducing the sizes of the $post$ vectors. We say that f is *redundant* in $(a, pre, post)$ if $(a, pre, post)$ is equivalent in terms of further probability computations to $(a, pre \setminus \{f\}, post)$. We note that f is redundant in $(a, pre, post)$ if each formula g guarding $(b, pre', post')$, *reachable from* a, is *conditionally independent* of f given pre' [26].

Following these observations, we designed an optimized algorithm that only adds a formula f to the memory if there exists some reachable formula conditionally dependent on it. Note that the order in which we add formulas to activities names, as well as the traversal order of the activities, affects the effectiveness of the optimization and the size of the resulting BP. Choosing the optimal order can be shown to be NP-complete. As a heuristic, we process the activities in a "top-down" manner, starting from the root. Our experiments in Section VII demonstrate the efficiency of our optimization.

Example 6.6: Figure 7 depicts the result of evaluating a TOP-1 query, given the bounded-memory c–likelihood function of table II. The "pre" and "post" vectors associated with each activity name are given in brackets, and contain a sufficient amount of information for computation of following $c-likelihoods$. For instance, as the likelihood of choosing a British Airways flight is dependent on the choice of search type, the search type ("Flights only") is recorded within the pre-condition vector of the $Flights''$ activity. Now, in order for it to appear in the pre-condition vector of the $\$Flights$ activity, it must also appear as a post condition for

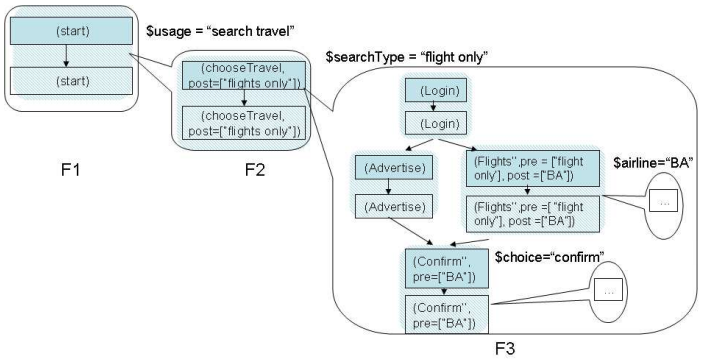

Fig. 7. TOP-1 BP for the bounded-memory c–likelihood function

the chooseTravel activity. Similarly, the pre-condition of the $\$Confirm$ activity contains only the choice of airline (as its guarding formulas are independent of the choice of search type, given the choice of airline).

Note that each condition added (either as pre- or post-condition) adds to the number of total specification activity names. The naive solution would consider all combination of guarding formulas over the compound activities, that is create 3^4 new activities out of each activity (3 possible choices for usage, search type, flights and hotels), that is a total of $3^4 * 8 = 648$ new activities. In contrast, our heuristics entails only 3 new activities for $chooseTravel$, 9 new activities for $Flights'$, $Flights''$, $Hotels$, $Confirm'$ and $Confirm''$ (each combination of pre (searchType) and post (Flight / Hotel / Confirmation) conditions), that is a total of 45 new activities, compared with the 648 of the naive algorithm.

C. Unbounded-memory c–likelihood functions

Last, we consider general c–likelihood functions that do not adhere to any of the previously discussed classes. For such c–likelihood functions we may show that computation of the $top-k(q,s)$ is impossible.

Theorem 6.7: The BEST-MATCH problem is undecidable, even if all activation-completion DAGs of the given BP have a simple, chain-like structure.

The proof (omitted here) is by reduction from the halting problem of a Turing Machine, known to be undecidable.

Multiple expansion sequences*:* To conclude, we withdraw the assumption (from Sect. III) of a known total order on the activities expansion, explaining the required adjustments.

- The c–likelihood function, previously defined as $\delta(e,f)$ for an EX-flow e and a formula f should now be defined as $\delta(e,n,f)$ with n being a node of e and f being a guarding formula of an implementation of n's activity. $\delta(e,n,f)$ is the likelihood of n being chosen for expansion, and its chosen expansion being guarded by f.
- A single execution flow may now be obtained through multiple expansion orders. The f–likelihood of a flow e (Definition 3.4) is now defined as f–likelihood $(e) = \sum_{e'|e'\to e}$ f–likelihood $(e')\times$ c–likelihood (e',n',f) where n' is the node of e' expanded to form e and f guards the corresponding implementation of n'.
- The definitions of c–likelihood function classes considered above are correspondingly adjusted, where dependencies relate to both the choice of a node to be expanded as well as the implementation chosen.

VII. Experiments

To validate our approach we have implemented the algorithms and tested their performance. Recall that our evaluation algorithm consists of two steps: in the first all matches are computed and in the second the TOP-K out of these are retrieved. As mentioned in section V, the algorithm for finding all matches is an adaptation of the algorithm of [2]. This algorithm was extensively analyzed and experimented in [2], and was shown to be efficient and scalable. The same result apply here, hence we do not discuss it in this section. Consequently, we focus on step 2, retrieval of top-k EX-flows of a given specification, under various settings.

Unfortunately, there are no standard benchmarks which could be used here. Existing benchmarks for process management systems are targeted to test the server performance, its implementation, correctness, etc. The process specifications they use are too small and simplified for our needs [27], [28]. Obtaining real-life statistics on EX-flows of commercial Web applications was also impossible, as the data is valuable to their operators and kept confidential. We thus used a dedicated synthetic benchmark of our own. For all algorithms, we have presented in the previous sections theoretical complexity bounds for their execution time. The goal of our experiments was to check if the behavior they expose in practice sheds some new light on these bounds. The experiments were run on Pentium4 3.0GHz, 1GB RAM and Windows XP Professional.

We start by considering top-k analysis in presence of a memory-less c–likelihood function. We have varied five parameters: the average number of implementations for each activity (denoted below $imp\#$), the average size (number of activities) of the implementation graphs (denoted $impSize$), the average nesting depth of the process, i.e. the number of nested distinct activities that can be invoked before hitting a recursion (denoted $depth$), the overall number of activities in the BP, and the number k. The c–likelihood function over the possible implementations was randomly generated.

The first set of experiments tested the sensitivity of the top-k analysis to the number of activity implementations in the analyzed BP (that is, the BP representing all matches to the query). A representative sample of the results is depicted in Figure 8(a). We vary here $imp\#$ from 1 to 1000, while keeping $impSize$, $depth$, the overall number of activities, and k constant (50,15,10000 and 10 resp.). The figure shows the execution time (in seconds) as a function of $imp\#$. We observe a linear growth that is due to the fact that for every activity, all of its possible implementations are examined. Note that even for a very large $imp\#$, the computation time is feasible (7 seconds for 1000 possible implementations for each of the 10000 activities).

The second set of experiments tested the sensitivity of the algorithm to the overall number of activities in the BP. A sample of the results is depicted in Figure 8(b). Here we keep $impSize$, $depth$, $imp\#$ and k constant (50,15,50 and 10 resp.)

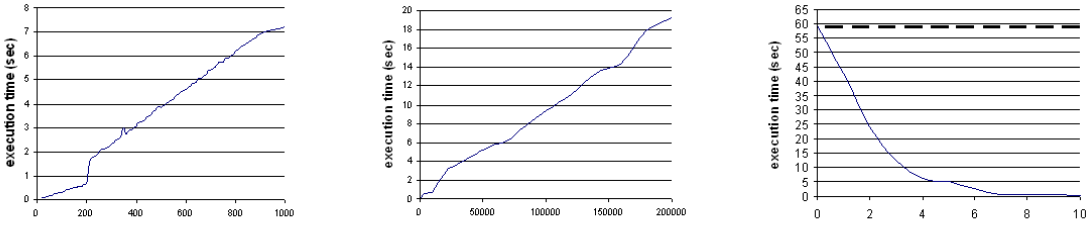

Fig. 8. (a) $imp\#$ (b) activities number (c) memory gap

and vary the activities number from 0 to 200000. The figure shows the execution time as a function of the activities number. The growth is linear even though the theoretical bound of the algorithm was quadratic, as this quadratic complexity is in fact a multiplication of the number of activities and the number of implementations for each compound activity, which explains the linearity in both. Here too, the results demonstrate that large BPs with hundreds of thousands of different activities can be processed in feasible execution time.

We also varied $impSize$ (resp. $depth$) while keeping the other parameters constant. This showed no significant effect on the execution time. Intuitively, this change only induces a different order of accessing the activities, but does not change the essence of computation. Finally, we varied k while keeping the other parameters constant. The execution time showed again a linear growth. Indeed, `TOP-K` computes the top-k flows in an incremental manner (the computation of the j'th most likely flow utilizes the previously computed $j-1$'th most likely flows), consistent with this linear behavior.

We next examined our analysis in presence of a bounded-memory c–likelihood function. As the algorithm uses the algorithm for memory-less functions, analyzed above, it reacts to changes of the above parameters in a similar way. Recall that the worst case complexity of the algorithm is polynomial in the size of the all-matches specification, with the exponent determined by the sum of the activities memory bounds. Our proposed optimization aimed at lowering the exponent to the actual minimal required memory. To evaluate the optimization effectiveness we compared the running time of the naive and optimized variants, as a function of the difference (memory gap) between the actual required memory and the memory bound. A representative example is depicted in Fig. 8(c), showing performance graphs for the basic algorithm (dashed line) and for the optimized algorithm (full line). The data used is an all-matches BP with 10000 activities and $depth = 15$, and naive memory bound of size 10, varying the memory gap from 0 to 10. Observe that the execution time of the basic algorithm stays constant with respect to changes in the actual memory, as it considers only the memory bound; in contrast, the optimized algorithm performs significantly better as the actual required memory drops (i.e. the memory gap grows), exemplifying the effectiveness of our optimization.

VIII. Conclusion

This paper studies, for the first time, the problem of top-k query evaluation over BPs. We have studied the complexity of the problem for various classes of likelihood functions, and presented efficient algorithms for query evaluation whenever possible. Important applications of our results include adjusting web-sites design to the needs of certain user groups, personalization of on-line advertisements, focusing on particular target audience, and enhancing business logic. We intend to study such applications as future research.

Acknowledgement: The research has been partially supported by the European Project MANCOOSI and the Israel Science Foundation.

References

[1] "Business Process Execution Language for Web Services," http://www.ibm.com/developerworks/library/ws-bpel/.
[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] D. Deutch and T. Milo, "Type inference and type checking for queries on execution traces," in *Proc. of VLDB*, 2008.
[5] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. on Knowl. and Data Eng.*, vol. 16, no. 9, 2004.
[6] R. Silva, J. Zhang, and J. G. Shanahan, "Probabilistic workflow mining," in *KDD*, 2005.
[7] D. Deutch and T. Milo, "Querying structural and behavioral properties of business processes," in *Proc. of DBPL*, 2007.
[8] N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *Proc. of VLDB*, 2004.
[9] P. Sen and A. Deshpande, "Representing and querying correlated tuples in probabilistic databases," in *ICDE*, 2007.
[10] N. Friedman, L. Getoor, D. Koller, and A.Pfeffer, "Learning probabilistic relational models," in *Proc. of IJCAI*, 1999.
[11] S. Sanghai, P. Domingos, and D. Weld, "Dynamic probabilistic relational models," in *Proc. of IJCAI*, 2003.
[12] S. Abiteboul and P. Senellart, "Querying and updating probabilistic information in xml," in *Proc. of EDBT*, 2006.
[13] B. Kimelfeld and Y. Sagiv, "Matching twigs in probabilistic xml," in *Proc. of VLDB*, 2007.
[14] S. P. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability*. Springer-Verlag, 1993.
[15] T. Oates, S. Doshi, and F. Huang, "Estimating maximum likelihood parameters for stochastic context-free graph grammars," in *ILP*, 2003.
[16] B. Courcelle, "The monadic second-order logic of graphs," *Inf. Comput.*, vol. 85, no. 1, 1990.
[17] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*. Springer-Verlag, 1992.
[18] K. Etessami and M. Yannakakis, "Algorithmic verification of recursive probabilistic state machines," in *TACAS*, 2005.
[19] M. Reynolds, "An axiomatization of pctl," *Inf. Comput.*, vol. 201, no. 1, 2005.
[20] T. Brazdil, A. Kucera, and O. Strazovsky, "On the decidability of temporal properties of probabilistic pushdown automata," in *Proc. of STACS*, 2005.
[21] D. Grigori, F. Casati, M. Castellanos, M. U .Dayal, and M. Shan, "Business process intelligence," *Computers in Industry*, vol. 53, 2004.
[22] J. Eder, G. E. Olivotto, and W. Gruber, "A data warehouse for workflow logs," in *Proc. of EDCIS*, 2002.
[23] F. Casati, M. Castellanos, N. Salazar, and U. Dayal, "Abstract process data warehousing," in *Proc. of ICDE*, 2007.
[24] D. Chamberlin, "Xquery: a query language for xml," in *Proc. of SIGMOD*, 2003.
[25] P. L. T. Pirolli and J. E. Pitkow, "Distributions of surfers' paths through the world wide web: Empirical characterizations," *World Wide Web*, vol. 2, no. 1-2, 1999.
[26] J. Pearl, *Causality*. Cambridge University Press, 2000.
[27] "activebpel," http://www.activebpel.org.
[28] Http://www.oracle.com/technology/bpel/index.html.