# Optimal Top-K Query Evaluation for Weighted Business Processes *

Daniel Deutch
Tel Aviv University

Tova Milo
Tel Aviv University

Neoklis Polyzotis
UC Santa Cruz

Tom Yam
Tel Aviv University

## ABSTRACT

A Business Process (BP for short) consists of a set of activities that achieve some business goal when combined in a flow. Among all the (maybe infinitely many) possible execution flows of a BP, analysts are often interested in identifying flows that are "most important", according to some weight metric. This paper studies the following problem: given a specification of such a BP, a weighting function over BP execution flows, a query, and a number $k$, identify the $k$ flows with the highest weight among those satisfying the query. We provide here, for the first time, a *provably optimal* algorithm for identifying the top-k weighted flows of a given BP, and use it for efficient top-k query evaluation.

## 1. INTRODUCTION

A Business Process (BP for short) consists of a set of activities which, when combined in a flow, achieve some business goal. BPs are typically designed via high-level specifications (e.g. using the BPEL standard specification language [3]) which are compiled into executable code. As the BP logic is captured by the specification, tools for querying and analyzing possible execution flows (EX-flows for short) of a BP specification are extremely valuable to companies [6].

A single BP typically induces a large (possibly infinite, for recursive BPs) set of possible EX-flows. Among all these EX-flows, analysts are often interested only in a subset that is relevant for their analysis. This subset is described via a query. Since the number of query answers (qualifying EX-flows) may itself be extensively large (or even infinite), it is important to identify those that are "most important", where the notion of importance is captured by some weighting metric that depends on the analysis goal. This paper considers the problem of finding, given a BP specification, a weighting metric over EX-flows, and a number $k$, the top-k weighted EX-flows of the specification. We provide here, for the first time, a provably optimal algorithm for identifying the top-k weighted EX-flows of a given BP; this algorithm is utilized for efficient top-k query evaluation.

For some intuition on the kind of BP analysis that one may be interested in, and the corresponding queries and weighting metrics, let us consider a simple example. Assume we are given a BP of a Web-based shopping mall with virtual shops of various vendors. A customer of the mall may be interested in buying a Toshiba TV and a DVD player of lowest overall price. Suppose that among the possible EX-flows containing a purchase of two such appliances [query], the lowest-priced one [weighting] is achieved when the user first subscribes to the Toshiba customer club. Such a result may suggest that subscription is beneficial even if it entails some registration fee. Alternatively, suppose that the same user is interested in minimal shipping time [another weighting]. In this case, the preferred flows may have both products purchased at the same store.

As another example, the Web-site owner may be interested to identify the most profitable/popular EX-flows [weighting] that lead to a TV purchase [query]. The answer here may be used, for instance, to target relevant personalized advertisements to users. Several weighting functions of interest may be combined to form a single weighting metric.

To formally study top-k query evaluation, we first propose a generic model for weighted EX-flows. To weigh EX-flows, we assume that each choice taken during the EX-flow bears some weight (denoted $cWeight$, for choice weight), and that $cWeight$s of choices throughout the EX-flow are aggregated to obtain the EX-flow weight (denoted $fWeight$, for flow weight). For example, $cWeight$ may be the price of the product chosen at a given point of the EX-flow, or the likelihood of a user clicking on a given store link. In the first case, summation may be used for aggregation; for likelihoods we may use multiplication. Following common practice [11], we require the aggregation to be monotonic w.r.t. the progress of the flow. This captures most practical scenarios, e.g., the total price of a shopping cart subset does not exceed the price of the full cart, even in the presence of discount deals.

It is important to note that the $cWeight$ of a given choice may vary at different points of the EX-flow and may depend on the course of the flow so far and on previous choices. For instance, the price of a given product may be reduced if the user had previously subscribed to a customer club or bought over two products from the same vendor; the likelihood of clicking on a certain store link may depend on stores previously visited. Thus $cWeight$ is modeled as a *function* whose

*This research was partially supported by the Israeli Ministry of Science Eshkol Grant, the Israel Science Foundation, the US-Israel Binational Science Foundation, a IBM faculty development award and NSF grant IIS-0447966


*Proceedings of the VLDB Endowment,* Vol. 3, No. 1

input includes not only the choice itself but also information about the history of the EX-flow thus far. We derive an equivalence relation *equiv* that captures how much the $cWeight$ of a given choice is affected by preceding choices, and provide a top-k algorithm whose worst-case complexity is polynomial in the count of classes in *equiv* and the size of the BP specification, and linear in the size of the output.

We then study the *optimality* properties of top-k algorithms in this setting. Following common practice [11], we employ the notions of *optimality* and *instance optimality*, that reflect how well a given algorithm performs compared to all other possible algorithms in its class. We show that the properties of the $fWeight$ function dictate the optimality of top-k algorithms in this setting: for general monotone $fWeight$ function, we show that no algorithm is optimal or even instance optimal. However, we further consider a class of *semi-strongly monotone* functions that arise frequently in practice. We show that still no optimal algorithm exists for such functions, but we show *instance optimality* for our algorithm. Finally, we show that for the further restricted class of *strongly monotone*, our algorithm is in fact *optimal*.

We conclude with an experimental study of our algorithm performance. Our experiments show that in all realistic scenarios, *including when the conditions that guarantee optimality are not met*, our algorithm outperforms the worst-case bound, as well as previously proposed algorithms, by an order of magnitude.

**Difficulties and novelty.** A first difficulty here is the *infinite* search space. While each individual EX-flow is finite, a single BP may have an *infinite* number of possible EX-flows (analogous to how a *grammar* defines an infinite set of *words*), due to the possibly recursive nature of BPs. Moreover, even for non-recursive BPs, the size of a single EX-flow may be exponential in the specification size. Indeed, standard $A^*$-style [5] search algorithms may fail to halt in this context and, moreover, become prohibitively inefficient, even for non-recursive BPs. Our previous work ([7]) has attempted to overcome these difficulties. As we show in this work, the performance of the solution proposed in [7] is unsatisfactory for several realistic scenarios. In contrast, the novel top-k algorithm presented here is *provably optimal*, and outperforms the algorithm of [7] *by over 90%* in real-life scenarios (see Section 5). Furthermore, the algorithm of [7] was tailored to *probabilistic* BPs and thus has limited applicability comparing to the novel algorithm which is *generically applicable to any weight metric*.

**Paper organization** Section 2 presents our model for weighted BPs and EX-flows. Section 3 presents our top-k algorithm, and Section 4 discusses its optimality. Section 5 describes our experiments. Section 6 concludes with related work. For space constraints, proofs are deferred to the Appendix.

# 2. PRELIMINARIES

We start by recalling the standard basic model for BPs and EX-flows [6], then extend it to support weighted BPs. As a running example, we will use the web-based Shopping Mall BP from the Introduction.

## 2.1 BP Specifications

At a high-level, a BP specification encodes a set of activities and the order in which they may occur. A BP specification is modeled as a set of node-labeled DAGs. Each DAG has a unique *start* node with no incoming edges and a unique *end* node with no outgoing edges. Nodes are labeled by activity names and directed edges impose ordering constraints on the activities. Activities that are not linked via a directed path are assumed to occur in parallel. The DAGs are linked through *implementation relationships*; the idea is that an activity $a$ in one DAG is realized via the activities in another DAG. We call such an activity *compound* to differentiate it from *atomic* activities having no implementations. Compound activities may have multiple possible implementations, and the choice of implementation is controlled by a condition referred to as the *guarding formula*.

We assume a domain $\mathcal{A} = \mathcal{A}_{atomic} \cup \mathcal{A}_{compound}$ of activity names and a domain $\mathcal{F}$ of formulas in predicate calculus.

DEFINITION 2.1. *A* BP specification *$s$ is a triple $(S, s_0, \tau)$, where $S$ is a finite set of node-labeled DAGs, $s_0 \in S$ is a distinguished DAG consisting of a single activity, called the* root*, $\tau : \mathcal{A}_{compound} \rightarrow 2^{S \times \mathcal{F}}$ is the* implementation function*, mapping each compound activity name in $S$ to a* set *of pairs, each consisting of an implementation (a DAG in $S$) and a guarding formula in $\mathcal{F}$.*

EXAMPLE 2.2. *Figure 1 shows an example BP specification. The root $s_0$ has precisely one activity named* `ShoppingMall`*. The latter has as its implementation the DAG $S_1$, which describes a group of activities comprising user login, the injection of an advertisement, the choice of a particular store, and the user exit (possibly by paying). The directed edges specify the order in which the activities may occur, e.g., a user has to login first before a store can be chosen. Some of the activities are not in a particular order, e.g., the injection of an advertisement and the choice of a store, which means that they occur* in parallel*.*

*Within $S_1$, we observe that* `Login` *and* `chooseStore` *are* compound *activities; the* `Login` *activity has two possible implementations $S_2$ and $S_3$ that are guarded by respective formulas. The idea is that exactly one formula is satisfied at run-time, e.g., the user either chooses to login with a Visa or a Mastercard credit card, and thus* `Login` *is implemented either by $S_2$ or $S_3$ respectively. A similar observation can be made for* `chooseStore`*, which has several possible implementations, but exactly one will be chosen at run-time depending on which guarding formula is satisfied. Note that the specification is recursive as $S_4$ may call $S_1$.*

We note that satisfaction of guarding formulas is determined by external factors, e.g. choices of the user or environment parameters. We assume that exactly one guarding formula can be satisfied when determining the implementation of a given compound activity occurrence. We observe that satisfaction of guarding formulas can change if activities occur several times. For instance, a user may choose to buy a "DVD" product the first time she goes through the activities of $S_4$, and a "TV" product the second time.

## 2.2 EX-Flows

An EX-flow is modeled as a nested DAG that represents the execution of activities from a BP. We model each occurrence of an activity $a$ by two $a$-labeled nodes, the first standing for the activity *activation* and the second for its *completion* point. These two nodes are connected by an edge. The edges in the DAG represent the ordering among activities activation/completion and the implementation relationships. To emphasize the nested nature of executions,

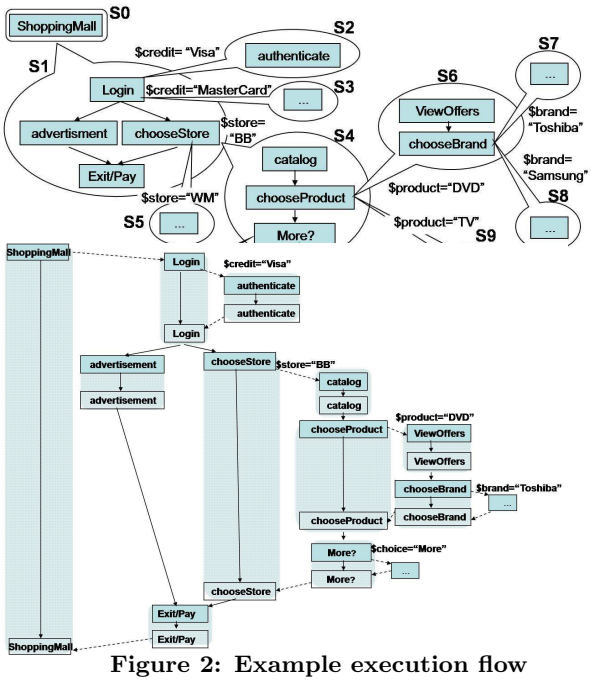


**Figure 2: Example execution flow**

the implementation of each compound activity appears in-between its activation and completion nodes. Of course, the structure of an EX-flow DAG must adhere to the structure of the corresponding BP specification, i.e., activities have to occur in the same ordering and implementation relationships must conform to function $\tau$.

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

- **Base EX-Flow:** *$e$ consists only of the activation and completion nodes of the root activity $s_0$ of $s$, connected by a single edge, or,*
- **Expansion Step:** *$e'$ is an EX-flow of $s$, and $e$ is obtained from $e'$ by attaching to some activation-completion pair $(n_1, n_2)$ of an activity $a$ in $e'$ some implementation $e_a$ of $a$, through two new edges, called* implementation edges, *$(n_1, \texttt{start}(e_a))$ and $(\texttt{end}(e_a), n_2)$, and annotating the pair with the formula $f_a$ guarding $e_a$.*

  *We require that this $(n_1, n_2)$ pair does not have any implementation attached to it already in $e'$, whereas all its ancestor compound activities in $e'$ do have one.*

  *In the attached implementation $e_a$, each activity node is replaced by a corresponding pair of activation and completion nodes, connected by an edge.*

  *We call $e$ an* expansion *of $e'$, denoted $e' \rightarrow e$.*

EXAMPLE 2.4. *An example EX-flow of the shopping mall BP is given in Figure 2. Ordering edges (implementation edges) are drawn by regular (resp. dashed) arrows. The EX-flow describes a sequence of activities that occur during the BP execution. The user logins with a* Visa *Credit Card, then chooses to shop at the* BestBuy *store. There, she chooses to look for a* DVD player *and selects one by* Toshiba*, then continues shopping at the same store, looking for a* TV*, and selects one also by* Toshiba *(this last part is omitted from the figure). Finally she exits and pays.*

We use $e' \rightarrow^* e$ to denote that $e$ was obtained from $e'$ by a sequence of expansions. An activity pair $a$ in an EX-flow $e$ is *unexpanded* if it is not the source of any implementation edge, implying that $a$ can be used to further expand $e$. We say that an EX-flow is *partial* if it has unexpanded activities, and *full* otherwise, and denote the set of all full flows of a BP specification $s$ by $flows(s)$. For a graph $e$, we say that $e$ is an EX-flow if it is a (partial or full) flow of some BP specification $s$.

To simplify the presentation, we impose a total order among unexpanded activities of any given partial flow, and assume that the expansion of activities follows this order. (This order can be achieved by a topological sort of the EX-flow DAG.) Thus, any partial EX-flow corresponds to a unique sequence of expansion steps from the base EX-flow. *This assumption is made solely for presentation considerations– all our results extend to a general context where multiple expansion orders are possible (see Appendix B).*

## 2.3 Weighted EX-flows

We assume an ordered domain $\mathcal{W}$ of weights. We use three functions: (1) $cWeight$ that describes the weight of each implementation choice, given a preceding sub-flow, (2) $aggr$ that aggregates $cWeight$ values, and (3) $fWeight$ that uses $cWeight$ and $aggr$ to compute flow weight.

***The cWeight function.*** Given a BP specification $s$, $cWeight$ is a partial function that assigns a weight $w \in \mathcal{W}$ to each pair $(e, f)$ where $e$ is an EX-flow of $s$ and $f$ guards the compound activity node of $e$ that is next to be expanded. Intuitively, the value $cWeight(e, f)$ is the weight of the implementation guarded by $f$, *given that $e$ is the flow that preceded it.*

EXAMPLE 2.5. *Re-consider the EX-flow in Fig. 2, and a $cWeight$ function assigning, to each implementation choice, the additional cost that it incurs to the customer. In this case $\mathcal{W}$ is the set of all positive numbers. The $cWeight$ of the choice $\$brand =$ "Toshiba", given that the preceding flow indicates that the product being purchased is a DVD, the shop is BestBuy and the user has identified herself as a Visa card holder, may be the Visa discounted price of a Toshiba DVD at BestBuy. The $cWeight$ of a Toshiba choice, given that the product is a TV and that the preceding flow already includes a purchase of a Toshiba product (the DVD), may reflect, e.g., a 20% discount for the second product. The $cWeight$ of other choices (like the store or the product type) may be zero, as they incur no additional cost to the user.*

***The Aggregation function.*** The weights along the EX-flow are aggregated using an aggregation function $aggr : \mathcal{W} \times \mathcal{W} \rightarrow \mathcal{W}$. The first input is intuitively the aggregated weight computed so far, and the second is the new $cWeight$ to be aggregated with the previous value. For instance, when computing purchase cost $aggr = +$ and $\mathcal{W} = [0, \infty)$; when computing likelihood, $aggr = *$ and $\mathcal{W} = [0, 1]$. We consider here aggregation functions that are associative, commutative, continuous and monotone, either increasing or decreasing (For formal definition of these properties see Appendix C). Observe that the aggregation functions $+$ and $*$, for cost and likelihood resp., satisfy the constraints.

***The fWeight function.*** Finally, the $fWeight$ of an EX-flow is obtained by aggregating the $cWeights$ of all choices made during the flow, and is defined recursively: if $e$ is an EX-flow consisting only of the root $s_0$, $fWeight(e) = 1_{aggr}$. Otherwise , if $e' \rightarrow e$ for some EX-flow $e'$ of $s$, then $fWeight(e) = aggr(fWeight(e'), cWeight(e', f))$, where $f$ is the formula guarding the implementation that is added to $e'$ to form $e$.

EXAMPLE 2.6. *Assume that a Toshiba TV and DVD, which individually cost 250$ and 150$ resp., are sold together with 20% discount (i.e. for 320$). The cWeight for the first choice of Toshiba TV (DVD) is 250$ (resp. 150$). The cWeight for the next choice, of DVD (TV), is 70$ (170$), computed as 320$ minus the cost already incurred for the first product. Using + for aggregation, cWeights along the flow are summed up, yielding the total 320$ deal price.*

Observe that when *aggr* is monotonically increasing (decreasing), so is $fWeight$, in the sense that the weight of an EX-flow increases (resp. decreases) as the execution advances. Generally, when $fWeight$ is increasing (as, e.g., for the overall price of purchases), we are interested in the bottom-k *full* flows (e.g. the cheapest overall price). When $fWeight$ is decreasing (as, e.g., for the likelihood of EX-flows), we are interested in top-k (e.g. the most likely) ones. Since all definitions and algorithms presented below apply symmetrically to both cases, we consider from now on only monotonically decreasing functions and top-k EX-flows.

***Top-k EX-flows.*** Given a BP specification $s$, a monotonically decreasing $fWeight$ function for $s$ and a number $k$, we study the problem of identifying the top-k weighted EX-flows in $flows(s)$ [1]. We refer to this problem as `TOP-K-FLOWS`. One may further consider queries, that select EX-flows of interest and in this case we retrieve the top-k out of the EX-flows matching the query. The common practice in such cases (see e.g. [7, 2]) is to employ a two-steps algorithm. In the first step we "intersect" the query $q$ with $s$, obtaining a new BP $s'$ whose flows are exactly those flows of $s$ that also match $q$. The second step is then to perform a top-k analysis over $s'$. An efficient algorithm for the first step was suggested in [6], and we thus focus here on the second step, i.e. `TOP-K-FLOWS`. For completeness, we recall in the Appendix D, (from [6]), the formal definition of queries and the details of this two step algorithm.

# 3. OPTIMAL TOP-K ALGORITHM

When devising an algorithm for `TOP-K-FLOWS`, one encounters two main difficulties. First, recall that BPs may be recursive, in which case the number of EX-flows to consider may be infinite. Second, even without recursion, the size of a single EX-flow may be exponential in the BP size, as a single graph may appear as the implementation of multiple compound activities in the flow. The conventional $A^*$ algorithm [5] may traverse the entire search space in the worst case, and thus fail to halt for recursive BPs. To observe that, consider the following simple example.

EXAMPLE 3.1. *Consider the following BP specification. Its root activity A has two possible implementations: the first, S1, guarded by a formula F1, consists of a single atomic activity a. The second, S2, guarded by F2, consists of a recursive invocation of A. The cWeight function is s.t. for every two EX-flows $e, e'$ $cWeight(e, F1) = cWeight(e', F1) = 0.5$ and $cWeight(e, F2) = cWeight(e', F2) = 0$; we use $aggr = *$. The algorithm will keep considering recursive expansions of A, each time obtaining EX-flows with decreasing weight, but nevertheless higher than 0, and will never terminate. We note that while in this example the cWeight values of some guarding formulas are 0, we may also design a (more complicated) weight function, with positive weights, for which the algorithm does not terminate.*

Moreover, $A^*$ can be inefficient (and incur EXPTIME in the BP specification size) even for non-recursive BPs, due to the sizes of materialized EX-flows (we provide such an example in Appendix E). To overcome these problems, we use the following two observations.

**Observation 1.** We first observe that some distinct nodes $n$, $n'$ may be in fact *equivalent*, in the sense that every sub-flow that may originate from $n$ may also originate from $n'$, having the same $fWeight$. In Example 3.1 above, all occurrences of nodes labeled by $A$ are equivalent. As another example, consider a weight function standing for product prices. If no deals are proposed, every two nodes $n, n'$ standing for the purchase of the same product $P$, are equivalent. If there are deals, then price of $P$ may depend on the flows histories preceding $n$ and $n'$. If these histories are the same in terms of purchasing the same products specified in the combined deal with $P$, then $n$ and $n'$ are equivalent. Equivalence is thus a relation between two pairs of (EX-flow,next-to-be-expanded-node). Before formally defining this relation, we introduce the auxiliary notion of (isomorphic) *sub-flows*: given an EX-flow $e$ and an activity $n$ of $e$, the sub-flow of $e$ rooted at $n$ consists of all nodes and edges appearing on some path in-between the activation and completion nodes of $n$. An *isomorphism* between two (sub-)flows $e$ and $e'$ is a one-to-one and onto matching between the nodes and edges of $e$ and $e'$, respecting node labels, guarding formulas and the edge relation.

DEFINITION 3.2. *Given two pairs of (EX-flow,next-to-be-expanded-node), $(e, n)$ and $(e', n')$, we say that $(e, n)$,$(e', n')$ are* equivalent *if (1) $n$ and $n'$ are labeled by the same activity name, and (2) for all EX-flows $\hat{e}$, $\hat{e'}$ s.t. $e \rightarrow^* \hat{e}$, $e' \rightarrow^* \hat{e'}$, and in which the sub-flows rooted at $n$ and at $n'$ (denoted $\hat{e_n}$ and $\hat{e'_{n'}}$ ) are isomorphic, $fWeight(\hat{e_n}) = fWeight(\hat{e'_{n'}})$.*

We denote the set of equivalence classes, for the given BP specification, by *equiv*. We assume in the sequel that *equiv* is known and finite, and explain below how to generate it.

**Observation 2.** We observe that existence of equivalence classes in combination with the monotonicity of $fWeight$ facilitates incremental-style computation.

LEMMA 3.3. *For every equivalence class $eq \in equiv$ and a compound activity node $n \in eq$, the following hold:*
*(1) there exists a best ranked (top$-1$) EX-flow originating at $n$ that contains no occurrence of any other node $n' \in eq$.*
*(2) for $j > 1$, there exists a $j$'th ranked flow originating at $n$ such that for any occurrence of a node $n' \in eq$ in it, the sub-flow rooted at $n'$ is one of its top $j-1$ flows.*

Let us illustrate the implications of these two observations.

EXAMPLE 3.4. *Re-consider example 3.1, and recall that while trying to retrieve the top-1 EX-flow rooted at A, the $A^*$-like algorithm has encountered a recursive invocation of A, and has examined possible EX-flows of the latter, thus resulting in an infinite loop. Following Lemma 3.3, this is redundant: to compute the top-1 EX-flow one may avoid considering flows that contain a recursive call to A (note that all occurrences of A, regardless of the flow preceding them, are equivalent). The top-2 flow may contain a recursive invocation of A, but the only sub-flow that needs to be*

[1]Certain EX-flows may have equal weights, which implies that there may be several valid solutions to the problem, in which case we pick one arbitrarily.

*considered as potential expansion for this occurrence of A is the (already computed) top-1 flow, and so on.*

`TOP-K` ***Algorithm.*** Following the above observations, we define an EX-flows table, $FTable$, (compactly) maintaining the top-k (sub)flows for each equivalence class. It has rows corresponding to equivalence classes, and columns ranging from 1 to $k$. Each entry contains a pointer to the corresponding sub-flow. In turn, every implementation of a compound activity node in this sub-flow is not given explicitly, but rather as a pointer to another entry in $FTable$, and so on. This guarantees that the size of each flow representation is bounded by the table size, avoiding the blow-up of EX-flow sizes. In what follows, *every EX-flow is represented via a single pointer to an entry at FTable* (we discuss the explicit construction of flows below).

The algorithm then operates in two steps. First, it calls a subroutine `FindFlows` which computes a compact representation of the top-k EX-flows within $FTable$, then it calls `EnumerateFlows` that uses the table to explicitly enumerate the EX-flows from this compact representation. We next explain the operation of these two subroutines.

***FindFlows.*** The `FindFlows` procedure maintains two priority queues $Frontier$ and $Out$ of (partial) EX-flows, ordered by $fWeight$. At each step, $Frontier$ contains all flows that still need to be examined. Upon termination, $Out$ will contain the top-k flows. Initially, $Out$ is empty and $Frontier$ contains a single partial EX-flow, containing only the BP root. At each step, we pop the highest weighted flow $e$ from $Frontier$. If $e$ is a full (partial) flow, the algorithm invokes $HandleFull$ ($HandlePartial$) to handle it.

**Algorithm:**HandlePartial
**Input**: e
1 $v \leftarrow getNext(e)$ ;
2 $TableRow = FTable.findEquivClass([e, v])$ ;
3 **if** $TableRow = NULL$ **then**
4 insert $[e, v]$ into $FTable$ ;
5 $Expansions \leftarrow AllExps(e)$ ;
6 **foreach** $(e', F') \in Expansions$ **do**
7 $r_{e'} \leftarrow aggr(r_e, cWeight(e, F'))$ ;
8 insert $(e', r_{e'})$ into $Frontier$;
9 **end**
10 **end**
11 **else**
12 $UnhandledExp \leftarrow \{e'_v \in TableRow \mid e'_v$ was not chosen for $(e, v)\}$ ;
13 **if** $UnhandledExp = NULL$ **then**
14 insert $((e, w_e), v)$ into $OnHold$ ;
15 **end**
16 **else**
17 $e''_v \leftarrow top(UnhandledExp)$ ;
18 $e' \leftarrow$ expand $e$ by *pointing* $v$ to $e''_v$;
19 $w_{e'} \leftarrow aggr(w_e, fWeight(e''_v))$ ;
20 insert $(e', w_{e'})$ into $Frontier$;
21 **end**
22 **end**

**Algorithm 1**: HandlePartial

***HandlePartial.*** `HandlePartial`, depicted in Algorithm 1, is given a partial flow $e$ and considers all possible expansions $e'$ of $e$. To that end, we assume the existence of an $AllExps$ function that allows to retrieve, given a partial flow $e$, all of its expansions (i.e. all $e'$ s.t. $e \rightarrow e'$), along with their weights. The algorithm first retrieves the next-to-be-expanded node $v$ of $e$ (line 1), and looks up its equivalence class in $FTable$ (line 2). If no entry is found, it means that we haven't encountered yet an equivalent node during the computation. We thus create a new row in $FTable$ for this equivalence class (line 4). Entries in this row will be filled later, when corresponding full flows are found. Then, we obtain all expansions of $e$ (Line 5), and for each such expansion we compute its $fWeight$ value, and insert it to the $Frontier$ queue for processing in the following iterations (Lines 6-9). Otherwise, if the appropriate row already exists in the table, we consider the partial EX-flows that appear in this row but were not yet considered for expanding $e$ (line 12). If no such EX-flow exists, (although the table entry exists), it means that $e$ was previously reached when expanding some other node $v'$ (which appears in $e$ as well). Following observation 2, we may compute the next best EX-flow without further expanding $e$. Thus, we put $e$ on hold (line 14). It will be released upon finding a full flow originating in $v'$ (see below). If an unused EX-flow exists, we take the highest ranked such EX-flow and "attach" it to $v$, that is, we make $v$ point to this flow (lines 17-18). We now compute the weight of the obtained EX-flow (line 19) and add it to $Frontier$ (line 20).

***HandleFull.*** `HandleFull` is depicted in Algorithm 2. First, the given full EX-flow $e$ is inserted into $Out$ (line 1). If $Out$ already contains $k$ flows, then we terminate. Otherwise, every node appearing in $e$, along with its preceding sub-flow (lines 7-9) define an equivalence class, used as entry at $FTable$. The sub-flow rooted at the node is then inserted into the table at that entry, if it does not appear there already (lines 10-11). Last, all EX-flows that were put by `HandlePartial` "on hold" due to a node participating in $e$, are returned to Frontier (lines 13-14).

**Algorithm:**HandleFull
**Input**: $e$, $w_e$
1 insert $(e, w_e)$ into $Out$ ;
2 **if** $|Out| = k$ **then**
3 Output $Out$;
4 **end**
5 **else**
6 **foreach** *node* $n \in e$ **do**
7 $e_n^{pre} \leftarrow$ the sub-flow of $e$ preceding $n$ ;
8 $e_n^{rooted} \leftarrow$ the sub-flow of $e$ rooted at $n$;
9 $w_n^{rooted} \leftarrow fWeight(f_n^{rooted})$ ;
10 **if** *not* $(e_n^{rooted} \in FTable)$ **then**
11 $FTable.update([n, e_n^{pre}], e_n^{rooted})$ ;
12 **end**
13 **foreach** $(e', n) \in OnHold$ **do**
14 insert $e'$ into $Frontier$ ;
15 **end**
16 **end**
17 **end**

**Algorithm 2**: HandleFull

***Explicit enumeration.*** When $FindFlows$ terminates, the top-k flows are compactly represented in $Out$ via pointers to entries in $FTable$; implementations of compound activities within the graphs appearing in these entries are possibly represented by pointers to other entries, etc. $EnumerateFlows$ follow these pointers to materialize EX-flows in $Out$. This

pointer chasing terminates, following observation 2.

We may prove the following theorem.

THEOREM 3.5. *Given a BP s (with cWeight and aggr) and a number k, if |equiv| is finite, the time complexity of Algorithm* `FindFlows` *is polynomial in |s|, k and |equiv|.*

***Generating** equiv.* Recall Def. 3.2, and observe that *equiv* is dictated by the fragment of the partial flow preceding a given choice, that in fact affects its $cWeight$. This fragment is termed as *memory*. For instance, if $cWeight$ stands for product prices, then the memory for a given product choice $P$ includes the choice of shop, and the previous purchase of products suggested in a combined deal with $P$. Thus, in practice, *equiv* may be derived e.g. from a database of product prices and proposed deals. $|equiv|$, and consequently the complexity of `TOP-K`, is then polynomial in $|s|$, with the exponent depending on the required memory size. Using a similar construction to that of [7], we may show the necessity of the exponential dependency on memory size (unless $P = NP$), and that the problem becomes undecidable when the memory size is unbounded (see Appendix A). In practice, the memory size for navigation choices in a web-site is typically bounded and very small (approx. 4 [17]).

# 4. OPTIMALITY PROPERTIES

We next consider the optimality of `TOP-K`. We start by defining our optimality measures, then analyze `TOP-K`'s optimality for weight functions with different properties.

## 4.1 Optimality Measures

We introduce the class of algorithms against which we compare `TOP-K`, the cost metric used for comparison, and the notions of optimality and instance-optimality.

***Competing algorithms.*** We define the class $\mathcal{A}$ of all *deterministic* correct top-k algorithms operating on the same input as `TOP-K` and having no additional information. An algorithm in $\mathcal{A}$ may retrieve an EX-flow by multiple calls to $AllExps$. It may obtain the $cWeight$ of each expansion choice and can apply $aggr$ to compute $fWeight$ of EX-flows, but cannot use information not obtainable in this manner.

***Cost metric.*** We consider the number of calls to $AllExps$ as the dominant computational factor, indicating the number of distinct (sub-)flows examined. The cost of an algorithm $A$ executed over an input instance $I$ (denoted $cost(A, I)$) is thus defined as the number of calls it makes to $AllExps$.

***Optimality and Instance Optimality.*** Following [11], we use two notions of optimality (within the class $\mathcal{A}$): $A \in \mathcal{A}$ is *optimal* if for each algorithm $A' \in \mathcal{A}$ and an input instance $I$ $cost(A, I) \leq cost(A', I)$. $A$ is *instance-optimal* if there exist constants $c, c'$ such that for each $A' \in \mathcal{A}$ and instance $I$, $cost(A, I) \leq c * cost(A', I) + c'$.

## 4.2 Optimality Results

We show next that the (instance) optimality of our algorithm is influenced by properties of the $fWeight$ function.

***Strongly monotone fWeight.*** We say that $fWeight$ is *strongly monotone* if for every two distinct (partial or full) flows $e, e'$, we have $fWeight(e) \neq fWeight(e')$. In particular, this implies that the weight strictly decreases as the flow advances. We prove the following.

THEOREM 4.1. *For strongly monotone $fWeight$ functions,* `TOP-K` *is optimal within $\mathcal{A}$.*

PROOF SKETCH. The proof works by contradiction and is based on the following lemma (proved in the Appendix):

LEMMA 4.2. *Given some input $I$, let $e_{term}$ be the worst solution in Out upon termination. Whenever an $e \neq e_{term}$ is popped from Frontier, $fWeight(e) > fWeight(e_{term})$.*

Now, if an algorithm $A$ expands, on some input $I$, less nodes than `FindFlows`, there exist a flow $e$ and a node $v \in e$ s.t. `FindFlows` expanded $v$ but $A$ has not expanded any node equivalent to $v$ (`FindFlows` expands at most a single node of each equivalence class). Denote $fWeight(e_{term})$ by $w_{term}$. $fWeight(e) > w_{term}$ by the lemma. Consider some $w^*$ s.t. $aggr(fWeight(e), w^*) > w_{term}$ (such $w^*$ exists as $aggr$ is continuous), and design an input $I'$, similar to $I$ but with a new implementation of $v$'s activity with $cWeight$ of $w^*$. $A$ is wrong on $I'$: it acts on $I'$ as on $I$, never seeing $v$, thus missing an EX-flow having a better weight than $w_{term}$. □

***Semi-Strongly Monotone fWeight.*** In a realistic setting, some choices do not incur any change to $fWeight$. (For instance, in our shopping mall example, the choice of store or product type induce a zero added cost). Consequently, some flows may share the same $fWeight$ value. Still, the number of such flows sharing any specific $fWeight$ value, is typically bounded. To model this we define, for each constant $c$, the notion of *c-strongly monotone $fWeight$*:

DEFINITION 4.3. *For a constant c, an $fWeight$ function is c-strongly monotone for a BP specification s, if for every weight $w$, $|\{e \in flows(s) \mid fWeight(e) = w\}| \leq c$.*

The following theorem holds.

THEOREM 4.4. *For every constant c, and every BP specification along with a c-strongly monotone $fWeight$ function,* `TOP-K` *is instance optimal within $\mathcal{A}$.*

The proof is similar to that of Theorem 4.1, and is given in Appendix A. Note that *no* optimal algorithm is possible here, as follows:

THEOREM 4.5. *No algorithm within $\mathcal{A}$ is* optimal *for all c-strongly monotone $fWeight$ functions.*

PROOF SKETCH. The proof assumes the existence of an optimal algorithm $A$, then defines an algorithm $A'$ with a different order for the expansion of equally weighted EX-flows, and constructs an input for which the order chosen by $A'$ yields less expansions before finding the top-1 flow. The full details are given in Appendix A. □

***Weakly monotone fWeight.*** Finally, we consider the (not so common in practice) case of weakly monotone $fWeight$ functions. Here users may perform an unbounded number of consecutive choices that incur no change to the EX-flow weight. Unfortunately, in this case our algorithm is not (instance) optimal, but we can show that in this case *no* (instance) optimal algorithm exists.

THEOREM 4.6. *No algorithm within the class $\mathcal{A}$ is (instance) optimal for all weakly monotone $fWeight$ functions.*

## 5. EXPERIMENTAL STUDY

We present an experimental study of our algorithm based on synthetic and real-life data. The study evaluates the performance of the algorithm in practice relative to the worst-case bounds implied by our analysis, examining cases where optimality is guaranteed as well as cases where it is not.

Note that TOP-K gradually fills in $FTable$, and halts once it discovered the top-k flows. We implemented a variant of TOP-K, termed WC (for worst-case), that fills in all entries of FTable before terminating, and compared the performance of TOP-K to WC. Performance-wise, WC is similar to the algorithm in [7] which also computes the top-k flows rooted at *every* activity. A comparison of TOP-K to WC thus provides a comparison to [7], demonstrating the significant performance gains achieved by our new algorithm.

We have implemented the algorithm in C++ and ran our experiments on a Lenovo T400 laptop, with Intel Core2 Duo P8600 processor and 2GB RAM. We ran two series of experiments. First, we used synthetic data to study the algorithm performance and scalability. Second, we used real data, in the context of the Yahoo! Shopping Web-site, to evaluate the performance in a real life setting.

*Experiments with Synthetic Data.* We generated our synthetic input by varying a number of different parameters that affect the complexity of the TOP-K-FLOWS problem. The ranges of parameter values were chosen based upon surveys on the structure of typical Web Applications [13, 17]; to examine the scalability of our techniques, we favored values that are on the higher end of the spectrum. In what follows, we describe the parameters and the chosen value ranges.

**BP Specifications size.** We varied the total number of activities in our BP specifications from 1000 to 40000; we note that [13] states that a typical number of activities in a given Web-site is 4000. To demonstrate scalability we have studied here BP specifications whose size is up to 10 times larger than that of a typical application.

**History Bound.** We considered bounded-history $cWeight$ functions with bounds ranging from 0 to 10. A previous study [17] on the behavior of Web surfers concluded that a typical history bound is 4.

**Equivalence classes.** Recall that the number of activities in the BP together with the size of the history bound determine the number of equivalence classes, and thus also the size of the $FTable$. The values above yielded BPs with 1K-260K different equivalence classes.

**Monotonicity strength.** We varied the percentage of $cWeights$ that are equal to the neutral value of the aggregation function. This percentage determines how strongly/weakly monotone is the $fWeight$ function, and in turn, to what extent conditions that guarantee optimality hold (see Section 4). To further study the effect of monotonicity strength on the algorithm performance, we varied the *standard deviation* of the distribution of $cWeight$ values (we considered uniform and normal distributions); high standard deviation implies greater monotonicity strength.

**Number of results.** We varied $k$ in the range 1 to 500.

**Additional Parameters.** As for the BP structure (i.e. number of activities in each implementation graph and number of possible implementations for each activity), we tried parameter values that are up to 5 times greater than observed in the real-life case of Yahoo! Shopping Web-site: we varied the number of implementation choices for each activity ranging from 2 to 1000, and the number of activities in each implementation from 100 to 1000. But given a fixed number of equivalence classes, we noted no significant effect of the BP structure on the algorithm performance, and consequently, we show the results for a fixed number (namely 50) of possible implementation choices (the number of activities in each implementation is dictated by the overall BP size and the number of possible implementation choices).

A representative sample of the experimental results is presented below. Figure 4(a) examines the execution times (in seconds) of TOP-K and WC for increasing number (in thousands) of equivalence classes. (The scale for the time axis in all graphs is logarithmic). Since our experiments showed that the shape of the BP graphs and the history bound do not affect the performance (given a fixed number of equivalence classes), we show here one representative sample where the history bound is 5. The number $k$ of requested results here is 100. (We will consider varying $k$ values below). The figure shows the performance of TOP-K for $cWeight$ values in the range [0,1] with different distributions. This includes uniform and normal distributions with average value of 0.5 and varying standard deviation of 0.2, 0.1, and 0 (the latter corresponding to all-equal $cWeight$ values). WC always fills in all entries of the $FTable$, thus is not sensitive to the $cWeight$ distribution, and we show only one curve for it.

Compared to WC, TOP-K generally shows an improvement of 90-99%, positively correlated with the variance of the distribution of $cWeight$ values. This is because variance in $cWeight$s implies variance in the EX-flows $fWeight$, exploited by the greedy nature of TOP-K which quickly separates the top-k results from the rest. As we shall see below, such variance of $cWeight$s is indeed common in real-life BPs. In the extreme (unrealistic) case where all $cWeight$ values are identical, i.e. standard deviation 0, the performance of WC and TOP-K became the same (as the early stop condition does not hold, and the flows table must be fully filled), thus we show only the WC curve.

Figure 4(b) examines the execution times of WC and TOP-K for a growing number $k$ of requested results (for the same distributions of $cWeight$s as above). The number of equivalence classes here is 200K and the history bound is 5. We can see that the running time increases only moderately as $k$ grows, with TOP-K steadily showing significantly better performance than WC and exhibiting similar behavior to what have been described above. (The increase for WC is less visible in the graph due to the logarithmic time scale).

Figure 4(c) examines the effect of the monotonicity strength of the weight function, on TOP-K's execution time. We fix $k$, the number of equivalence classes, and the history bound (to 100, 40K, and 5, resp.), and vary the percentage of neutral weights, with the non-neutral weights uniformly distributed. At the left-most end, there are no neutral weights and TOP-K is guaranteed to be optimal; at the right-most (very unlikely) case all weights are neutral, and TOP-K and WC exhibit the same execution times (as the flows table must be fully filled). We see that the performance of TOP-K is significantly superior even when the conditions for optimality do not necessarily hold. In particular, in all realistic scenarios where less than 90% of the weights are neutral, TOP-K improves over WC by more than 75%.

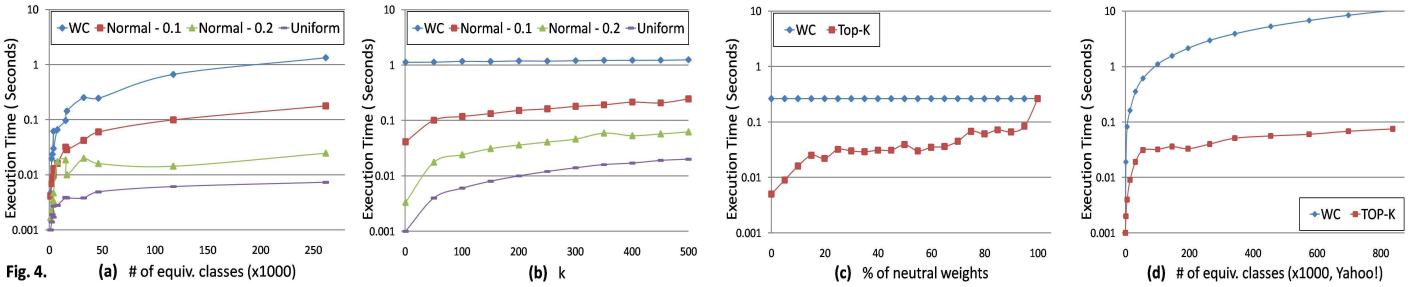


Fig. 4.

*Experiments with Real Data.* Our second set of experiments was performed using over (part of) the Yahoo! Shopping Computer Store [20]. We used real data – products details, pricing information (including deals, reductions, etc.), and more – obtained from the site through a Web interface offered by Yahoo!. The obtained BP specification consists of 5976 activities with an average of 2.6 implementation choices per compound activity and a history bound of 4, yielding approx. 840K equivalence classes. The variance in $cWeight$ values (costs) of choices for each compound activity (product type) is high, e.g. the average RAM price is 192$, with a standard deviation of 510$.

We considered increasingly large parts of the BP specification (corresponding to the outcome of evaluating decreasingly selective queries in the evaluation process depicted at the bottom of Section 2). Fig. 4(d) depicts results for 15 representative such subsets, involving increasing counts of equivalence classes–the leading factor in the performance of the `TOP-K` algorithm. At the extreme right, all equivalence classes participate in the computation. Observe that `TOP-K` outperforms `WC` by a factor of over 98%, demonstrating scalability and good performance.

Note that this is also an example for a case where the $A^*$-like top-k algorithm does not halt: the BP specification contains a mutual recursion that is due to the *back* button facilitated by the web-site. As pressing the back button incurs no cost, this recursive choice has a 0 weight, leading to a case similar to that depicted in Example 3.1. In contrast, our optimal top-k algorithm not only halts but shows an extremely good performance.

## 6. CONCLUSION AND RELATED WORK

We considered in this paper top-k query evaluation in the context of weighted BPs. We analyzed different classes of weight functions and their implications on the complexity of query evaluation, and have given, for the first time, a *provably optimal* algorithm for identifying the top-k EX-flows of BPs. We showed that our algorithm outperforms previous work [7] by an order of magnitude. The top-k algorithm of [7] was used in [8] as a *subroutine*, for evaluation of projection queries over probabilistic BPs. Replacing this sub-component by the faster, optimal algorithm given in the present work, will yield the same acceleration that we had observed here, for the evaluation of projection queries.

Top-k queries were studied extensively for relational and XML data [12]. Notably, [11] presented an instance-optimal algorithm for top-k queries that aggregate individual scores given to joining tuples. In our context, one may think of the $cWeight$ as the equivalent of an individual score, and of $fWeight$ as the aggregation of $cWeight$ values along a given EX-flow. Difficulties specific to our settings are that (1) the size of a given flow, thus the number of aggregated scores, is unbounded (2) the particular properties of the cWeight functions are unique to EX-flows and (3) the number of items (EX-flows) that are ranked is infinite. Note that while an infinite setting also appears in *streamed* data analysis [16], such works aggregate over a *bounded size sliding window*, whereas we study aggregation over flows of unbounded size.

Ranking by likelihood was also studied in for *Probabilistic Databases* (PDBs) [18] and *Probabilistic XML* [1, 15], extending relational databases and XML, resp., to a probabilistic setting. For example, [18] and [15] study top-k query evaluation over PDBs and Probabilistic XML, resp. In contrast to relational data and XML, our model for BP flows allows representation of an *infinite* number of items, out of which the top-k are retrieved. Works on *Probabilistic process specifications* (e.g. [14, 10, 4]) either suffer from low expressivity or incur infeasibility of query evaluation. Analysis of non-weighted processes in the context of verification (rather than top-k analysis) was discussed in e.g. [9].

Finally, we note a complementary line of works (e.g. [19]) on the optimization of ETL processes, that allow to construct a repository of execution flows that occurred in the *past* (and consequently, to derive weight functions such as choices likelihoods to be used within our model). This is complementary to our analysis of possible *future* executions.

Extending our algorithms to more powerful models and query features, including e.g. value-based joins, projection, negation and non-monotone weight functions, while preserving low complexity is a challenging future research.

## 7. REFERENCES


[1] S. Abiteboul and P. Senellart. Querying and updating probabilistic information in xml. In *Proc. of EDBT*, 2006.
[2] Active XML. http://activexml.net/.
[3] Business Process Execution Language for Web Services. http://www.ibm.com/developerworks/library/ws-bpel/.
[4] B. Courcelle. The monadic second-order logic of graphs. *Inf. Comput.*, 85(1), 1990.
[5] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *JACM*, 32(3), 1985.
[6] D. Deutch and T. Milo. Type inference and type checking for queries on execution traces. In *Proc. of VLDB*, 2008.
[7] D. Deutch and T. Milo. Evaluating top-k queries over business processes (short paper). In *Proc. of ICDE*, 2009.
[8] D. Deutch and T. Milo. Top-k projection queries for probabilistic business processes. In *Proc. of ICDT*, 2009.
[9] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In *PODS*, 2006.
[10] K. Etessami and M. Yannakakis. Algorithmic verification of recursive probabilistic state machines. In *TACAS*, 2005.
[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4), 2003.
[12] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.
[13] T. Jones. *Estimating Software Costs*. McGraw-Hill, 2007.
[14] J. Kemeny and J. Snell. *Finite Markov Chains*. Springer, 1976.
[15] B. Kimelfeld and Y. Sagiv. Matching twigs in probabilistic xml. In *Proc. of VLDB*, 2007.
[16] N. Koudas and D. Srivastava. Data stream query processing: A tutorial. In *Proc. of VLDB*, 2003.
[17] P. L. T. Pirolli and J. E. Pitkow. Distributions of surfers' paths through the world wide web: Empirical characterizations. *World Wide Web*, 2(1-2), 1999.
[18] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.
[19] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Qox-driven etl design: reducing the cost of etl consulting engagements. In *SIGMOD '09*, 2009.
[20] Yahoo! shopping. http://shopping.yahoo.com/.

# APPENDIX

# A. ADDITIONAL PROOFS

PROOF. (Lemma 3.3)

1. Let $e$ be a top-1 flow rooted at $n \in eq$. If $e$ contains no other node $n' \in eq$, we are done. Otherwise, consider the sub-flow $e'$ of $e$ that is the implementation of $n'$ (i.e. $e'$ appears in between the activation and completion nodes of $n'$). Since $n'$ and $n$ are equivalent, we may $e'$ is also an implementation of $n$. Furthermore, the weights appearing along $e'$ stay the same when $e'$ is rooted at $n'$ intact (again, due to the equivalence of $n$ and $n'$). Now, $e'$ is a sub-flow of $e$, and thus $fWeight(e') \geq fWeight(e)$ due to monotonicity. Thus, $e'$ is also a top-1 flow rooted at $e$. If there are additional nodes $n'' \in eq$ still appearing in $e'$, we may repeat this process to omit them.
2. As for part (2) of the theorem, assume by induction that it holds for the $i$'th ranked flow rooted at $n$, for every $i < j$ (part (1) is the base case for this induction). Specifically it means that there exists some set of top-(j-1) flows rooted at $n$ for which no node $n' \in eq$ in it is the root of a sub-flow that its $j-1$-ranked flow. Now, if $e$ is a $j$'th ranked flow originating at $n$ bearing a node $n' \in eq$ in it, such that the sub-flow rooted at $n'$ is not one of its top $j-1$ flows, we may replace the sub-flow rooted at $n'$ by by its $j$–1 ranked flow, without decreasing weight of $e$.

□

PROOF. (Theorem 3.5)

The number of entries in $FTable$ is $k* \mid equiv \mid$. Now, for each flow node $v$ considered during the course of the algorithm execution, either it already appears in $FTable$, or it does'nt. The case where the sub-flow requested for $v$ does not appear in the table may only happen $k* \mid equiv \mid$ times, while computing the top-k flows rooted at $v$. the cost of computation for such cases is $O(\mid equiv \mid)$ for searching the table, (assuming that we have an index that allows, in O(1) time, to get the last (worst ranked) entry for a given row; otherwise there may be an additional factor of $k$) and then $O(1)$ of further computation - considering direct expansions of $v$, a total of $O(k* \mid equiv \mid^2)$.

If the sub-flow considered for $v$ does already appear in $FTable$, we only need to point the implementation of $v$ to the sub-flow that were already computed ($O(1)$). We next consider the number of times that this scenario may occur.

We start by considering the computation of top-1 flows. Now, consider some equivalence class $e$. Say that we've encountered some node $n \in e$, and then, before we are done computing the top-1 flow rooted at n, we have encountered, at another point of the search tree, another node $n' \in e$ . The course of the algorithm execution follows Observation 2 of Section 4: it suspends the computation for the top-1 flow of $n'$, until computation of the top-1 flow of $n$ is done (by putting $n'$ "on hold", line 14 of Algorithm `HandlePartial`. The number of such suspensions, while computing the top-1 flow of $n$, is bounded by the size of the specification $s$, for each such $n \in e$ and for each $e$. The number of such equivalence classes is $\mid equiv \mid$. The same argument holds for computation of the $i$'th highest weighted flow, for each $i = 1, ...k$, leading to a total bounded by $\mid s \mid^2 * \mid equiv \mid *k$ for this case. The total complexity is thus polynomial in $\mid equiv \mid$, $k$, and $\mid s \mid$. □

*Undecidability.* We may show that `TOP-K-FLOWS` is undecidable in general (if $|equiv|$ may be infinite), as stated in the text following Thm. 3.5.

PROOF. The proof is by a reduction from the *halting problem*. Given a Turing Machine M, the idea is to "encode" $M$ using the BP specification. The states of $M$ are represented by activity names; implementations model the transitions between states, as well as changes to the tape and to the head location; and the history of flow is utilized to allow "read" operations from the tape.

More formally, given a Turing Machine with a set of states $Q$, an initial state $q_0 \in Q$, an accepting state $q_F \in Q$, a tape alphabet $\Gamma$ and a transition function $delta$, we generate a BP specification whose set of compound activity names is $Q$, and additionally it contains an atomic activity $a$ .The implementation set of each compound activity corresponding to a state $s$, contains a single-node implementation for each activity name $s'$ (possibly $= s$) such that there is a transition from $s$ to $s'$ according to $\delta$. The cWeight of such transition is intuitively 1 if the transition is legitimate, according to $\delta$ and to the current symbol under the head, and 0 otherwise. As $cWeight$ function is unbounded-history, it is allowed to determine its value according to the entire preceding flow: this flow determines uniquely the tape state and the head location. For the accepting state, its single implementation consist of the atomic activity $a$, with $cWeight$ of 1. We use multiplication for aggregation, and seek for full flows with $cWeight$ higher than 0. □

*NP-hardness.* We may further show that `TOP-K-FLOWS` is NP-hard in the required memory size, may not be solved in PTIME, as stated in the text following Thm. 3.5.

PROOF. We use a reduction from Set Cover. For simplicity, we consider here a monotonically increasing $fWeight$ function and bottom-k computation. ($aggr$ is the + function, and $\mathcal{W}$ are positive numbers). But the proof works symmetrically for monotonically decreasing $fWeight$ and top-k. (Simply take negative $cWeight$s instead of the positive ones we use below). Given an instance of set cover, namely a set $X = \{X_1, ..., X_n\}$ of items, a set of subsets $S = \{S_1, ..., S_m\}$ and a bound $B$, we construct a BP as follows: its activity names are $R$ (root), $S_1, ..., S_m$ (compound) and $a$ (atomic). Each $S_i$ ($i = 1, ...m$) bears $2 * m+1$ implementations: for each $j = 1, ...m$, $S_i$ has two implementations, each consisting of a single node whose activity is $S_j$: the first is guarded by a formula "$\$S_i$ = chosen", and the second by "$\$S_i$ = not chosen"; the last implementation of $S_i$, guarded by "$\$S_i$ = done", consists of $a$. The $cWeight$ of "$\$S_i$ = chosen" and "$\$S_i$ = not chosen" is 0, and that of "$\$S_i$ = done" depends on the last $B$ choices: it is 1 if and only if the set of $S_i$'s for which "$\$S_i$ = chosen", within this set of $B$ choices, covers $X$. Otherwise, its $cWeight$ is 3. We use addition for $aggr$. There exists a flow of $fWeight$ smaller than 2 if and only if there exists a set cover of size smaller than $B$.

□

*Proof of Theorem 4.1.* We give here a full proof of Theorem 4.1 (a sketch of which appears in the paper body).

In the following, we use $w_e$ to denote the $fWeight$ of a flow $e$.

PROOF. (Theorem 4.1)

We start by showing an invariant satisfied by Algorithm `FindFlows`.

*Lemma 4.2.* Given some input $I$, let $e_{term}$ be the worst solution that appears in $Out$ upon termination. Whenever an $e \neq e_{term}$ is popped from $Frontier$ (in Line 4 of Algorithm `refinedFindFlows`), $fWeight(e) > fWeight(e_{term})$.

PROOF. Denote $fWeight(e_{term})$ by $w_{term}$. Also, for a flow $e \neq e_{term}$, denote $fWeight(e)$ by $w_e$. By the definition of strong monotonicity, there exists a unique flow $e_{term}$ with weight $w_{term}$. If this flow $e_{term}$ already appear in the frontier when $e$ is popped from it, then clearly $w_e > w_{term}$. If it does not appear yet in $Frontier$, then $Frontier$ must contain at least some "prefix" $e'$ of $e_{term}$, (i.e. some flow $e'$ s.t. $e' \rightarrow^* e_{term}$). Here too $w_e > w_{e'}$, since $e$ was popped out and not $e'$, and by the strong monotonicity, we also have $w_{e'} > w_{term}$. Thus, $w_e > w_{e'} > w_{term}$. □

We next use the lemma to prove Theorem 4.1 assume that there is a sample instance $I$ and an algorithm $A$ such that $cost(A, I)$ is less , i.e. $A$ calls $AllExps$ a smaller number of times. Since $A$ and our algorithm produce the correct solution on input $I$, it must hold that they output the same weight for the last (worst weighted) result in $Out$.

It is easy to show that for each equivalence class $E$, Algorithm `FindFlows` calls $AllExps$ at most once for any node $v \in E$. As assumed, Algorithm $A$ invokes $AllExps$ less. Thus, there exists at least one equivalence class $E$ such that Algorithm `FindFlows` expanded some node $v \in E$, but Algorithm $A$ did not expand $v$ or any node equivalent to it. Say that $v \in e$, where $e$ is some specific flow. As $e$ was considered by Algorithm `FindFlows`, Lemma 4.2 guarantees that $w_e > w_{term}$. Thus, there exists at least one partial flow $f$ for which expansion of some of its nodes were considered by Algorithm `FindFlows` and were not considered by Algorithm $A$. Define $w^*$ such that $aggr(w_e, w^*) > w_{term}$.

As $aggr$ is **continuous**, and $w_e > w_{term}$, there exists such $w^*$. We construct another input instance $I'$ as follows. The $AllExps$ function is the same for all the nodes expanded by $A$, as well as the $cWeight$ function and the aggregation function.

For a node $v$, the $AllExps$ function returns a single implementation $e'$ consisting of a single atomic activity, and the corresponding $cWeight$ is $w^*$. As for the subsequent $m$ compound activities in $e$ if exists, we design $w_1, ..., w_m$ such that *given $e$ as history*, $aggr(w_e, w^*, w_1, ...w_m) > w_{term}$. Now the corresponding flow has a weight higher than $w_{term}$ and should have been added to $Out$. But Algorithm $A$ did not, and thus commits a mistake. This contradicts our assumption that $A$ is a correct algorithm, and thus it must hold that Algorithm `FindFlows` is optimal for input $I$. Given that $I$ is an arbitrary input, it follows that Algorithm `FindFlows` is optimal.

PROOF. (Theorem 4.4)

The proof works by contradiction. Let $c$ be the bound on the number of consecutive expansions of any flow $e$ that lead to a flow $e'$ with $fWeight(e) = fWeight(e')$. Assume that $A$ is better than Algorithm `FindFlows` by a factor greater than $c$. Let $w_{term}$ be as defined in the proof of Theorem 4.1 above. Note that for *every* flow $e$ considered by `FindFlows`, $fWeight(e) \geq w_{term}$. There are at most $c$ flows $e$ such that $fWeight(e) = w_{term}$. Other than these $c$ flows, for all other flows $e'$ considered by `FindFlows`, $fWeight(e') > w_{term}$. Consequently, there exists at least one flow $e$ that was inserted by `FindFlows` to frontier, such that $fWeight(e) > w_{term}$, and $f$ was not considered by $A$. The proof then proceeds similarly to the proof of Theorem 4.1 above. □

PROOF. (Theorem 4.5) By contradiction, let us assume the existence of some optimal algorithm $A$. Given a number $n$, we construct a BP specification $s$ as follows: its activities are $A_1, ..., A_n$ (compound) and $a$ (atomic). $A_1$ is the root activity.

Again, we construct the BP gradually, each time obtaining an intermediate BP, executing $A$ on it, and changing the BP according to the prior execution of $A$. $A_1$ has two possible expansions, one containing only $A_2$ and the second containing only $A_3$. We continue to construct the specification according to the behavior of $A$. If $A$ chooses to expand $A_2$ (resp. $A_3$), then $A_3$ (resp. $A_2$) will have as implementation the single activity $A_4$. $A_3$ ($A_2$) and $A_4$ have a single implementation, consisting of $A_5$; for $i = 5, ..., n$ $A_i$ a single implementation, consisting of $A_{i+1}$, and $A_n$ has a single implementation consisting only of $a$.

We use multiplication for $aggr$, and a $cWeight$ function that assigns 1 to every guarding formula, apart from that guarding the implementation of $A_4$, which is weighted 0.5. The induced $fWeight$ is semi-strongly monotone. When executed over $s$, $A$ expands $n-1$ nodes (it expands 3 nodes out of the first 4: $A_1$, either $A_2$ or $A_3$, and $A_4$, then the rest $n-4$ nodes). In contrast, a different algorithm that chooses a different order of expansion for this instance, will only expand here $n-2$ nodes (not going through $A_4$) while still being always correct. □

PROOF. (Theorem 4.6)

Given an algorithm $A$, we construct a BP $s$ as follows: its activities are $r$ (root), $A_1, ..., A_n$ (compound) and $a$ (atomic). We use multiplication for $aggr$, and a $cWeight$ function that assigns 1 to every guarding formula. Clearly, $aggr$ is weakly monotone. We construct the BP gradually, each time obtaining an intermediate BP, executing $A$ on it, and changing the BP according to the prior execution of $A$.

The root activity $r$ has two implementations, the first consisting of only $A_1$ and the second of only $A_2$. At first, we set the implementations of both $A_1$ and $A_2$ to be $a$, and execute $A$ on this (partial) BP. We then examine which activity was expanded first by $A$ - $A_1$ or $A_2$. Since $A \in \mathcal{A}$, this choice *does not depend on the implementations of $A_1$ and $A_2$*. Thus, we change $s$ as follows: if $A$ chose to expand $A_1$ first, we set the two implementations of $A_1$ to consist of $A_3$ and $A_4$ respectively, while the implementation of $A_2$ still consists of $a$. Next, $A$ has three choices for the next expansion: $A_2$, $A_3$, and $A_4$. For the chosen activity, we set its two implementations to consist of $A_5$ and $A_6$ respectively, while these of $A_2$ and $A_3$ consist of $a$. We repeat this process until obtaining a BP with $n$ compound activities, then allow the implementation of $A_n$ to consist of $a$ as well. We denote the obtained BP by $s$.

Clearly, when $A$ is executed over $s$, it expands $n$ nodes,

because each activity $A_i$ that it chose, such that $i < n$, only led to implementations containing further compound activities. In contrast, we claim that there exists an optimal algorithm $A'$ that, when executed over $s$, expands only $log(n)$ activities.

To observe that this is true, we show two lemmas, as follows:

LEMMA A.1. *There exists a flow $e_{short}$ in $flows(s)$ (with $fWeight = 1$, same as the $fWeight$ of any flow of $flows(s)$), that is obtained by $log(n)$ expansions.*

PROOF. Assume by contradiction that *every* flow in $flows(s)$ consists of more than $log(n)$ expansions of activities. Since every compound activity in $s$ has two implementations, each containing a distinct activity name, we obtain that there are over $2^{log(n)} = n$ distinct activities in $s$, in contradiction to the way $s$ was constructed. □

LEMMA A.2. *There exists a correct algorithm $A'$ that, when executed on $s$ finds $e_{short}$ as the top-1 flow, while expanding exactly the activity nodes appearing in $e_{short}$.*

PROOF. First, note that all $cWeight$ values within $s$ are identical; thus, for *every* order of expansion over the activities of $s$ (that comply with the specification), there exists a correct algorithm $A'$ (that is, $A'$ is correct for every input) that, when executed over $s$, follows this expansion order. Specifically, there is a correct algorithm that expands exactly the activities participating in $e_{short}$. □

As a corollary of these two lemmas, when evaluated over $s$, $A'$ expands at most $log(n)$ nodes, while $A$ expands $n$ nodes. $A'$ is thus better by a factor of $\frac{n}{log(n)}$. $n$ may be chosen as we wish, and given a constant $c$ we may choose $n$ such that $\frac{n}{log(n)} > c + c'$. Thus $A'$ is better than $A$ by a non-constant factor, and $A$ is not instance optimal.

This concludes the proof of Theorem 4.6.

# B. MULTIPLE EXPANSION SEQUENCES

We have assumed above (see the text following Example 2.4), for simplicity of presentation, the existence of a total order over the expansion of activities. We next withdraw this assumption, and explain the needed adjustments to our definitions and algorithms.

- The $cWeight$ function, previously defined as $cWeight(e, f)$ for an EX-flow $e$ and a formula $f$ should now be defined as $cWeight(e, n, f)$ with $n$ being a node of $e$ and $f$ being a guarding formula of an implementation of $\lambda(n)$. $cWeight(e, n, f)$ is the weight of $f$, given that $n$ was chosen for expansion in $e$
- $fWeight$ is now defined as $fWeight(e) = \max_{e' | e' \to e} aggr(fWeight(e'), cWeight(e', n', f))$ where $n'$ is the node of $e'$ expanded to form $e$ and $f$ is the guarding formula on the corresponding expansion of $n'$
- Given a partial flow $e$, Algorithm `TOP-K` now examines the expansions of each node of $e$, rather than just those of a single node that is next-to-be-expanded in $e$.

Our results all extend to the settings of multiple expansion sequences.

# C. CONSTRAINTS OVER THE AGGREGATION FUNCTION

We provide the exact definition of the constraints imposed over the aggregation function $aggr$, described intuitively in section 2.

1. $aggr$ is associative and commutative, namely for each $x, y, z \in \mathcal{W}$, $aggr(aggr(x, y), z) = aggr(x, aggr(y, z))$, and $aggr(x, y) = aggr(y, x)$
2. $aggr$ is *continuous*, that is for each $x, y, z \in \mathcal{W}$, if $aggr(x, y) < aggr(x, z)$ then there exists $w \in \mathcal{W}$ such that $aggr(x, y) < aggr(x, w) < aggr(x, z)$
3. $aggr$ has a neutral value, denoted $1_{aggr}$. Namely for each $x \in \mathcal{W}$, $aggr(x, 1_{aggr}) = aggr(1_{aggr}, x) = x$
4. $aggr$ is monotonically increasing or decreasing over $\mathcal{W}$. Namely, either $\forall s, x, y \in \mathcal{W} \quad x \geq y \implies aggr(s, x) \geq aggr(s, y)$ and $aggr(s, x) \geq s$, or the same for $\leq$.

# D. QUERIES

We quote here the formal definitions for queries and their matches. These definitions appear also in [6]. We then consider top-k query evaluation.

DEFINITION D.1. *We say that a DAG $e$ is an* abstract flow *if there exists some BP $s'$ s.t. $e \in flows(s')$. An* execution pattern*, abbr. EX-pattern, is a pair $p = (\hat{e}, T)$ where $\hat{e}$ is an abstract EX-flow whose nodes are labeled by labels from $\mathcal{A} \cup \{\text{ANY}\}$ and may be annotated by guarding formulas. $T$ is a distinguished set of activity pairs and edges in $\hat{e}$, called* transitive.

EXAMPLE D.2. *An example query (EX-pattern) is given in Fig. 3. The double-lined edges (double-boxed nodes) are* transitive *edges (activities). The query looks for EX-flows where the user chooses a DVD of brand Toshiba (possibly after performing some other activities, corresponding to the transitive edges), then chooses also a TV (of any brand). The* `ShoppingMall` *activity is transitive indicating that its implementation may appear in any nesting depth;* `chooseProduct` *is not transitive, requiring the brand choice to appear in its direct implementation.*

Given a BP specification $s$, a query (EX-pattern) $p$ selects the EX-flows $e \in flows(s)$ that contain an occurrence of $p$. Intuitively, nodes and edges of the EX-pattern are matched to nodes and edges of EX-flows, respecting activity names, ordering and implementation edges. Formally,

DEFINITION D.3. *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 $\psi$ 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 labelled by* ANY *may be mapped to a node 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 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 $[n, m]$.*

An EX-flow $e$ belongs to the query result if there exists some embedding of $p$ into $e$. We then say that $e$ *satisfies* $p$

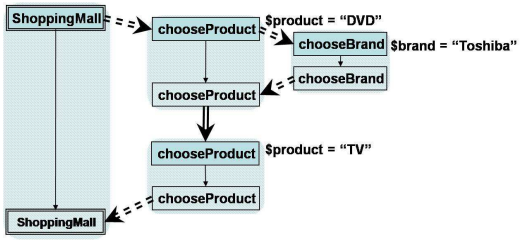


**Figure 3: Example query (EX-pattern)**

## D.1 Query Evaluation

We name TOP-K-ANSWERS as the problem of finding, given a BP specification $s$, weight function over its flows, *an EX-pattern* $p$, and a number $k$, the top-k flows of $s$ satisfying $p$. The following theorem holds:

THEOREM D.4. TOP-K-ANSWERS *may be solved in time polynomial in* $|s|$,$k$, *and* $|equiv|$ *(with the exponent depending on* $|p|$*) and linear in the output size.*

PROOF. We combine two algorithms, as follows:

1. The first algorithm is the query evaluation algorithm of [6] that, given a BP specification $s$ and an EX-pattern $p$, constructs a BP specification $s'$, including only those EX-flows of $s$ that matches $p$. Intuitively, $s'$ is the "intersection" of $s$ with $p$, obtained by considering all possible splits of the query into sub-queries, then matching these sub-queries to the DAGs in $s$.
2. The second algorithm is our TOP-K algorithm, that retrieves the top-k EX-flows of the constructed $s'$.

The complexity of the first algorithm is $|s|^{|p|}$ [6], and so is the maximal size of the resulting BP $s'$. The second step, as shown above, is then polynomial in the size of its input $s'$, and in $k$ and $|equiv|$, and is linear in the output size. □

To show that this exponential dependency on the size of the query is inevitable, we define the decision problem BEST-ANSWER, which tests, given a weighted BP specification $s$, a query $q$, some $k > 0$, and a threshold $t$, whether the top-1 flow in TOP-K-ANSWERS is of weight higher than $t$. The following theorem holds.

THEOREM D.5. BEST-ANSWER *is NP-hard in* $|q|$.

PROOF. We prove the NP-hardness using a reduction from 3SAT, as follows.

Given a Conjunctive Normal Form formula $F$, with variables $\{X_1, ...X_n\}$ we generate a specification and a query $(s, q)$, as follows: the idea is to create a compound activity associated with each variable of the formula. This activity has two different implementations: for all $i$, the implementations of $X_i$ are $BP_iTrue$ and $BP_iFalse$. The former contains all clauses that $X_i$ satisfies, and the latter contains all clauses that $\neg X_i$ satisfies. The query requires all clauses of the formula $F$ to appear.

To formally prove that the reduction is valid, we give the following lemma.

LEMMA D.6. *There exists an EX-flow in* $flows(s)$ *satisfying the query* $q$ *if and only if the formula* $F$ *is satisfiable.*

PROOF. Let $e$ be an EX-flow in $flows(s)$ satisfying $q$. $e$ was obtained by choosing a subset of compound activities for which $BP_iTrue$ is chosen as implementation, and another subset for which $BP_iFalse$ was chosen. These choices correspond exactly to a satisfying assignment - For every variable whose corresponding compound activity has as implementation the 'false' ('true') graph, assign 'false' ('true'). This is indeed an assignment, as every compound activity can only have exactly one of the 'true' or 'false' graphs as an implementation in $e$, and it is satisfying as every clause node appears in $e$. The truth value assigned to the variable corresponding to this compound activity thus satisfies this clause.

Conversely, let A be a satisfying assignment. The EX-flow obtained by choosing as implementation for each compound activity, its "true" graph if A assigns "true" to the corresponding variable, and its "false" graph if A assigns "false" to it. This is indeed an EX-flow in $flows(s)$, since A is an assignment. This EX-flow satisfies $q$ as every node clause appears at least once. This is due to the fact that the assignment A is satisfying, thus for each clause, there is at least one variable whose truth value causes the clause to be true. □

# E. EXAMPLE FOR EXPTIME BEHAVIOR OF THE $A^*$-LIKE ALGORITHM

We have suggested in Section 3 an algorithm based on the idea of $A^*$ and noted that it has two pitfalls: the first is non-termination for recursive BPs, and the second is possibly EXPTIME behavior, even for non-recursive BPs. Example 3.1 shows the first pitfall (non-termination), and we next provide an example for the second pitfall (EXPTIME for non-recursive BP specifications).

EXAMPLE E.1. *Let* $n$ *be an integer, and consider the following BP specification, whose activity names are* $r$ *(root),* $A_1, ...A_n$ *(compound),* $a$ *and* $b$ *(atomic). We will use integer weights addition for aggregating these weights. The root activity bears two possible implementations: the first (named* $S1$*, guarded by a formula* $F1$*) has a single activity node labeled* $a$*, and the second (named* $S2$ *guarded by a formula* $F2$*) has two activities, both labeled by* $A_1$*;* $A_1$ *bears an single implementation, with two activities both labeled by* $A_2$*, and so on. I.e.,* $A_i$ *bears an single implementation, with two activities both labeled by* $A_{i+1}$ *for* $i = 1, ..., i = n - 1$*. The single implementation of* $A_n$ *bears a single atomic activity node* $a$*.*

*All* $cWeight$ *values are dictated by the choice of implementation and independent of the preceding flow. The* $cWeight$ *value of* $F1$ *is* $n - 2$*, while the* $cWeight$ *values of all other formulas are* 1*.*

*When looking for the top-1 EX-flow, the* $A^*$*-like algorithm would generate the EX-flows of size* $\Theta(2^n)$ *obtained from subsequent implementation choices of the implementations containing* $A_i$*, before identifying that the top-1 flow is in fact obtained by choosing the* $S2$ *implementation of the root. In contrast, our TOP-K Algorithm, presented in Section 3, will compute the top sub-flow rooted at each activity only once (through its use of* $FTable$*, see algorithm description), avoiding the exponential blow-up.*