Query Results Maneuvers using Concept Hierarchies

Venkata Ramana Kumari.Ch [1], B. Sowjanya Rani [2]
[1]M Tech Student, Dept of CSE, Aditya Engineering College, Surampalem, Andhra Pradesh, India,
[2]Assistant Prof, M.Tech, Dept of CSE, Aditya Engineering College, Surampalem, Andhra Pradesh, India,

Abstract:
Search Query Results on large databases (Pub Med, HPSS, and NERSC) often return a large number of results of which only a small subset is relevant to the end user. To reduce this information overload Ranking and categorization were developed earlier. Efficient navigation through results categorization and annotations is the focus of this paper. In this paper, we present a new system based on online shopping that implement the Top-K algorithm. This system assists that on-line shoppers navigated in most effective paths based on their specified criteria and preferences. The suggestions are continually adapted to choices/decisions taken by the users while navigating. Earlier works expand the hierarchy in a predefined static approach, without stressing on navigation cost. We show experimentally that the system outperforms state-of-the-art categorization systems with respect to the user navigation cost. We present an experimental study that our algorithm outperforms state-of-the-art ranking systems with respect to the navigation flow.

Index Terms: *Pub Med, Top-k algorithm, Effective Navigation, Ranking, and Categorization etc.*

I INTRODUCTION

On-line shopping is extremely popular nowadays, with millions of users purchasing products in shops that provide a Web interface. It is common for on-line shops to offer a vast number of product Options and combinations thereof [4]. This is very useful but, at the same time, makes shopping rather confusing. It is often very difficult to and the specific navigation path in the site that will lead to an "optimal" result, best suiting the needs and preferences of the given user.

Consider for example an on-line store which offers various processors, screens etc; that allows users to assemble computers from a variety of component parts. Consider a user that is interested in buying a cheap Intel processor computer. Suppose user can get a good price by first registering to the store's customers club. After passing through some advertisement page that provides such members with discount coupons, and finally buying a certain set of components (including a certain Intel processor) that, when purchased together with the above coupons, yields the cheapest overall price. Clearly, the user might be interested in knowing this information if she is after the deal with the best price [9]. Alternatively, the user may prefer combinations where the delivery time is minimal, or may want to use the experience of others and view the most popular navigation paths.

We present here ShopIT (Shopping assistant); a system which suggests the most effective navigation paths based on preferences and specified criteria. When the user starts her navigation in the site, she may specify her constraints and her ranking function of interest, and have the system compute and propose (an initial set of) top-k ranked navigation flows[2], out of these conforming to the constraints. The user then continues her navigation taking into account the presented recommendations, but may also make choices different than those proposed by the system.

Several challenges arise in the development of such a system. First, the number of possible navigation flows in a given web-site is not only large but infinite, as users may navigate back and forth between pages. Hence, enumerating and ranking all relevant flows is clearly not an option. Second, it is critical to maintain a fast response time in order to provide a pleasant user experience. Finally, as explained above, the computation must be flexible and adaptive, to account for run-time user choices and our algorithm is optimal [2] and very efficient.

II TECHNICAL BACKGROUND

We provide in this section some background on our model for Web applications and for queries over such applications.

***Web-based Applications*:** Our model for Web applications, introduced in [1,6], abstractly models applications as a set of (nested) DAGs - Directed A cyclic Graphs - each intuitively is corresponding to a specific function or service[1,3]. The graphs consist of activities (nodes), and links (edges) between them, that detail the execution order of activities. Each activity is represented by a pair of nodes, the first standing as the activity's activation point and the second as its completion point. Activities may be either atomic, or compound. In the latter case their possible internal structures (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, link traversals, variable values, etc. These are captured by logical formulas (over the user choices, variable values, etc.) that guard each of the possible implementations. A Web-based application may be recursive, where an (indirect) implementation of some activity a contains another occurrence of a.

Navigation Flow: A navigation flow[6] (in a given application) corresponds to concrete choices of implementations for compound activities. For instance, a possible navigation flow in our computer store is one where the user first reviews the possible deals, then chooses to purchase an Intel Motherboard, subsequently cancels her choice, buys a discount coupon and selects a CPU by HP, etc. Note that the number of possible navigation flows may be extensively large even for relatively small-scaled applications.

Ranking: The rank[6] of navigation flows is derived using two functions, namely, cWeight and fWeight. Function cWeight assigns a weight to each single (implementation) choice within a flow, depending on the course of the flow thus far and the objective that the user wishes to optimize, e.g., monetary cost or likelihood. Function fWeight aggregates the per-choice weights in a single score for the entire flow.

Top-k query results: The users' search criteria[6] are modeled by queries. These use navigation patterns, an adaptation of the tree patterns, found in existing query languages for XML, to nested application DAGs [1]. Our top-k query evaluation algorithm gets as input the schematic representation of the application, the user query and the chosen ranking metric, and efficiently retrieves the qualifying navigation flows with the highest rank. The algorithm operates in two steps. First, it generates a refined version of the original application representation, describing only flows that are relevant to the user request. Then it greedily analyzes the refined representation to obtain the best-ranked flows. The choices made by the user throughout the navigation are modeled as additional constraints/relaxations to the original query, and an efficient adaptive evaluation technique is employed to update the query result.

III SYSTEM OVERVIEW

We give here a brief overview of the main components and their interaction. Figure 1 depicts the system architecture.

Store Model[6]: The abstract model of the on-line store application and its cWeight information are stored in the ShopIT database.

The first component, namely the application abstract model, was manually configured. Many Web-based applications are specified in declarative languages and then an automated extraction of their abstract model structure is possible. The products information, including compatibility relation in-between products, as well as additional parameters such as products cost, discount deals, shipment time etc. were automatically retrieved via a standard Web interface. The cWeight function was automatically derived to reflect this data.

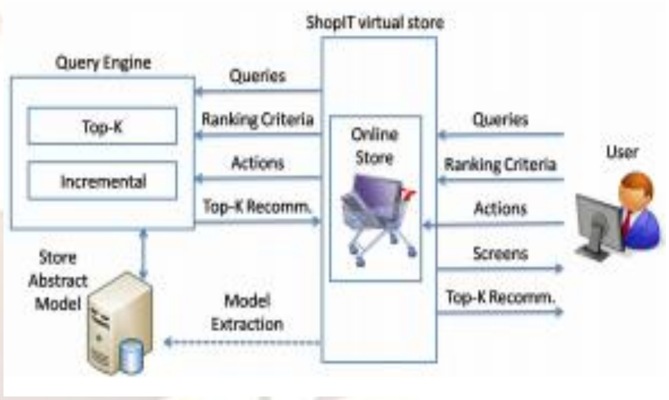

Fig 1: System Architecture

Query Engine: The query engine[6] is composed of two components. The first is the Top-k queries evaluator that receives, as input, from the user, her search criteria and chosen ranking metric, and computes the initial suggestion of top-k qualifying navigation paths. ShopIT supplies a Graphical User Interface that allows users to specify their criteria for search. The specified criteria are compiled into a navigation pattern, which in turn is evaluated over the Web Application model.

The second component is the adaptive recommendation engine that is continuously informed about the user actual navigation choices (or changes to her search criteria and ranking choice) and adapts the offered top-k suggestions accordingly. We have designed the query engine so that it is accessible through an API that allows the placement of user queries and preferences, and the retrieval of the corresponding recommendations. This general API can be used to incorporate ShopIT within a given website.

The ShopIT virtual store: Users interact with a virtual store[6] that wraps the original store. User actions are passed, through the API, to the ShopIT engine and to the (original) store application. The obtained recommendations are then presented to the user alongside with the resulting store screens. Each recommendation consists of a sequence of proposed actions, such as “click on button X”, “choose option Y at box Z”, etc., and is accompanied by its corresponding weight (e.g. total price, likelihood, etc.).

IV OPTIMAL TOP-K ALGORITHM

TOP-K Algorithm [2, 5]: we define an EX-flows table, F Table, (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 F Table, 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 F Table. 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 F Table, and then it calls Enumerate Flows that uses the table to explicitly enumerate the EX-flows from this compact representation. We next explain the operation of these two subroutines.
Find Flows: The Find Flows 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 Handle Full (Handle Partial) to handle it.

HandleFull: First, the given full EX-flow e is inserted into Out. If Out already contains k flows, then we terminate. Otherwise, every node appearing in e, along with its preceding sub flow defines an equivalence class, used as entry at F Table. The sub-flow rooted at the node is then inserted into the table at that entry, if it does not appear there already. Last, all EX-flows that were put by Handle Partial due to a node participating in e, are returned to Frontier.

```
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
```

HandlePartial: Handle Partial is given a partial flow e and considers all possible expansions e′ of e. To that end, we assume the existence of an All Exps function that allows to retrieve, given a partial flow e, all of its expansions (i.e. all e′ s.t. e → e′), along with their weights. The algorithm first retrieves the next-to-be- expanded node v of e, and looks up its equivalence class in F Table. 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 F Table for this equivalence class. Entries in this row will be filled later, when corresponding full flows are found. Then, we obtain all expansions of e, and for each such expansion we compute its fWeight value, and insert it to the Frontier queue for processing in the following iterations. 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. 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). We may compute the next best EX-flow without further expanding e. Thus, we put e on hold. It will be released upon finding a full flow originating in v′. 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. We now compute the weight of the obtained EX-flow and add it to Frontier.

```
Algorithm:HandlePartial
Input: e
$v \leftarrow getNext(e)$ ;
$TableRow = FTable.findEquivClass([e, v])$ ;
if $TableRow = NULL$ then
    insert $[e, v]$ into $FTable$ ;
    $Expansions \leftarrow AllExps(e)$ ;
    foreach $(e', F') \in Expansions$ do
        $r_{e'} \leftarrow aggr(r_e, cWeight(e, F'))$ ;
        insert $(e', r_{e'})$ into $Frontier$;
    end
end
else
    $UnhandledExp \leftarrow \{e'_v \in TableRow \mid e'_v$ was not chosen for $(e, v)\}$ ;
    if $UnhandledExp = NULL$ then
        insert $((e, w_e), v)$ into $OnHold$ ;
    end
    else
        $e''_v \leftarrow top(UnhandledExp)$ ;
        $e' \leftarrow$ expand $e$ by $pointing$ $v$ to $e''_v$;
        $w_{e'} \leftarrow aggr(w_e, fWeight(e''_v))$ ;
        insert $(e', w_{e'})$ into $Frontier$;
    end
end
```

V 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 F Table, 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 F Table before terminating, and compared the performance of TOP-K to WC that provides a comparison to [8]. A comparison of TOP-K to WC demonstrates the significant performance gains achieved by our new algorithm.

A representative sample of the experimental results is presented below. Figure 1(a) examines the execution times (in seconds) of TOP-K and WC for increasing number (in thou- sands) of equivalence classes. (The scale for the time axis in all graphs is logarithmic). 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.

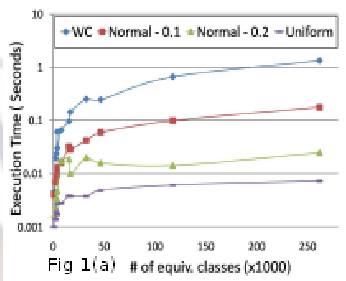

Fig 1(a)

Figure 1(b) examines the execution times of WC and TOP-K for a growing number k of requested results (for the same distributions of cWeights 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.

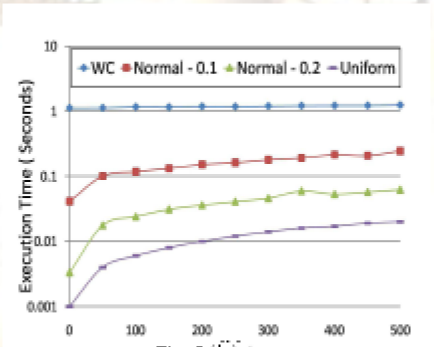

Fig 1(b)

Figure 1(c) examines the e®ect 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%.

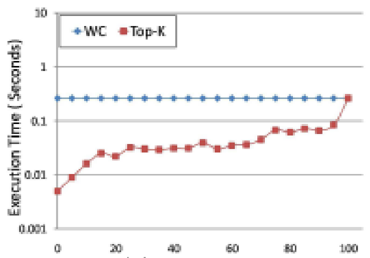

Fig 1(c)

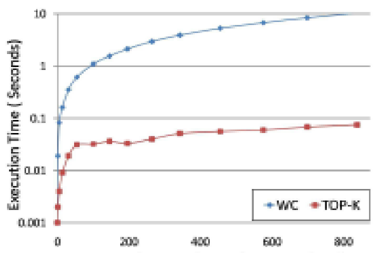

Fig 1(d)

Fig. 1(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.

VI CONCLUSION

Web-sites for on-line shopping typically offer a vast number of product options and combinations thereof. While this is very useful, it often makes the navigation in the site and the identification of the "ideal" purchase (where the notion of ideal differs among users) a confusing, non-trivial experience. This demonstration presents ShopIT (Shopping assistant), a system that assists on-line shoppers by suggesting the most effective navigation paths for their specified criteria and preferences. The suggestions are continually adapted to choices/decisions taken by the users while navigating. ShopIT is based on a set of novel, adaptive, provably optimal algorithms for TOP-K query evaluation in the context of weighted BPs (business process). 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 [5]. We showed that our algorithm outperforms by an order of magnitude.

VII REFERENCES

[1] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. "Querying business processes". In Proc. of VLDB, 2006.
[2] D. Deutch, T. Milo, N. Polyzotis, and T. Yam. "Optimal top-k query evaluation for weighted BPs"(submitted). 2009.
[3] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven web services. In PODS, 2006.
[4] Yahoo! shopping. http://shopping.yahoo.com/
[5] 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.
[6] Daniel Deutch, Tova Milo, Tom Yam," Goal-Oriented Web-site Navigation for On-line Shoppers".
[7] N. Koudas and D. Srivastava. "Data stream query processing: A tutorial". In Proc. of VLDB, 2003.
[8] D. Deutch and T. Milo. "Evaluating top-k queries over business processes (short paper)". In Proc. of ICDE, 2009.
[9] R. Dechter and J. Pearl. "Generalized best-first search strategies and the optimality of A*". JACM, 32(3), 1985.