PRO: Preference-Aware Recurring Query Optimization

Zhongfang Zhuang*, Chuan Lei†, Elke Rundensteiner*, Mohamed Eltabakh*
*{zzhuang, rundenst, meltabakh}@cs.wpi.edu, †chuan@nec-labs.com

*Computer Science Department
Worcester Polytechnic Institute
Worcester MA, 01609 USA

†NEC Labs. America
10080 N. Wolfe Road, SW3-350
Cupertino CA, 95014 USA

ABSTRACT

While recurring queries over evolving data are the bedrock of the analytical applications, resources demanded to process a large amount of data for each recurring execution can be a fatal bottleneck in cost-sensitive cloud computing environments. It is thus imperative to design a system responsive to users' preferences regarding how resources should be utilized. In this work, we propose PRO, a preference-aware recurring query processing system that optimizes recurring query executions complying with user preferences. First, we show that finding an optimal execution configuration is an NP-complete problem due to the cost interdependencies between consecutive executions. We propose an execution relation graph (ERG) model that effectively incorporates these dependencies between executions. This model enables us to transform our problem into a well-known graph problem. We then design a graph-based approach (called PRO-OPT) leveraging dynamic programming and pruning techniques with pseudo-polynomial complexity. Our experiments confirm that PRO consistently outperforms state-of-the-art solutions by 9 fold in processing time under a rich variety of circumstances on the Wikipedia datasets.

Keywords

Recurring Query; Preference-Aware; Execution Selection;

1. INTRODUCTION

Applications ranging from clickstreams analysis for online advertisement recommendations, log analysis for intrusion detection, to user sentiment inference on products are all recurring complex analytical tasks. These applications execute periodically on data subsets that arrive within the last hour(s), day(s), week(s), or even month(s) depending on the applications' scope of interest. Companies like Facebook, LinkedIn, and Twitter dominantly utilize in-house clusters with open source infrastructures to perform these tasks as recurring queries over massive evolving datasets [14, 15].

CIKM'16 , October 24-October 28, 2016, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/2983323.2983664

One key question for such in-house computing clusters is "*how to handle system overload?*" A system overload may incur severe delays in response time or even a failure in producing meaningful results when processing recurring queries with large amounts of evolving data. Thus, if a recurring query processing system cannot process every query execution within a reasonable latency, one feasible solution is to drop certain recurring query executions to assure success of all critical executions.

Dropping at the execution level produces exact results for a subset of executions, whereas tuple-based sampling techniques [2, 16] produce approximate results which are typically acceptable only for aggregation queries. Therefore, depending on the queries semantics and complexity, execution dropping can be the preferred solution.

However, execution dropping must consider users' preference as well as the integrity of results for analytical workloads. Thus, it is imperative to design a system that (1) allows data scientists to specify their execution selection preferences as part of the recurring query submitted to the system; (2) supports efficient recurring query processing with limited resources based on user preference functions; and (3) provides a proactive execution strategy for data fluctuations based on runtime statistics.

State-of-the-Art Techniques. Some prior work [10, 11] focused on supporting recurring queries. However, their solutions do not provide any system-level optimizations for application-driven preference requirements, nor do they skip executions to minimize the resource utilization.

Other systems [6, 12, 2, 16] minimize resource utilization by providing approximate or partial results for large scale analytical queries. More precisely, ApproxHadoop [6] explores input data sampling for workloads, while iMR [12] proposes to sample and load-shed at the data origination (i.e., where the data is generated). BlinkDB [2] aims to balance between result accuracy and response time requirements specifically for aggregation queries. G-OLA [16] supports tuple-based incremental online aggregation by partitioning data into smaller trunks of identical size on Spark.

None of these techniques target in particular recurring queries where future periodic executions of a query can be anticipated. Instead, their focus is limited to reduce the input size of *one individual execution* to manageable size of work. That is, their approximation techniques are designed to reduce the execution time at the tuple or data block granularity [6, 12, 2, 16]. In contrast, the preference-aware recurring query processing, in this work, requires a system to choose a subset of executions to complete (i.e., sampling at

the execution granularity) instead of sampling individual tuples/blocks from the data source. Such a system guarantees that all delivered results are a subset of the exact answer with consideration of user-specified preferences. This property is important since it allows the recurring application to rely on the results received to be correct.

Challenges. To achieve the design goals of preference-aware recurring query processing illustrated above, the following key challenges must be tackled.

Intractable Search Space of Execution Selection. Finding a subset of executions from all executions with the maximum satisfaction of user-specified preferences requires considering all possible combinations of query executions. This is a NP-complete combinatorial problem [13]. Plus, adding application preferences introduces additional dimensions to the problem.

Stochastic Execution Costs. Consecutive executions of a recurring query may share large segments of overlapping input data. Hence, a decision about one particular execution of a query may affect the costs of its subsequent executions and thus the final configuration. This *stochastic cost* nature of recurring query executions complicates our execution selection problem with uncertainties. Effective selection strategies must be designed to reduce the execution costs.

Contributions. Our contributions include:

• We establish a preference-aware recurring query model. This model incorporates the preference specification into PRO's optimization problem to ensure the maximum satisfaction of application preferences across the selected query executions (Sections 2 and 3).

• We show that our problem of finding a subset of executions that maximally satisfy user-specified preferences is NP-complete. Our platform-independent PRO optimizer is designed to utilize a dynamic programming strategy to generate an optimal execution plan for a given period of time with a pseudo-polynomial time complexity. Platform independence enables the PRO optimizer to be employed in any batch processing system, such as Hadoop, Spark, etc (Sections 4 and 5).

• Our experimental study on real-world datasets shows PRO consistently outperforms the state-of-the-art solution in processing time by 9 fold for various scenarios (Section 7).

2. PRELIMINARY

Recurring Query Parameters. A recurring query has three parameters, namely a window w, a slide s and a preference specification *pref*. The scope of data to be processed in each *execution* E_i of a recurring query Q is specified by the time window w on the dataset, while the slide s specifies the frequency of execution. Namely, two consecutive executions E_i and E_{i+1} apply the query Q to the data within time ranges $[t-w, t]$ and $[t+s-w, t+s]$ respectively, where t is the current timestamp. Lastly, $pref$ denotes the preference specification associated with the query Q.

The preference specification $pref$ has three parameters, including (1) a period of time in the form of m consecutive executions for a recurring query, (2) the number of executions k out of those m executions required by a user, and (3) a preference function $\mathcal{U}$ given by the user indicating the relative importance of each individual execution among the m executions for the application. We further explain the preference function in Section 3.1.

The system must guarantee that the number of executions dropped never exceeds the value of $m-k$. Given the limited available resources, we will drop executions such that the delivered answer is a subset of the original answer. Choosing the appropriate values for these three preference parameters requires domain knowledge from the users. For example, one way to determine k is to leverage the knowledge from domain experts on the freshness or significance of results. In this case, if the results slowly become stale, then a relatively small k can be chosen [5].

Under heavy workload, it may not be possible to remove excess load while still meeting the bound on k. In this case, we apply "admission control" on recurring queries, where the most costly queries whose preference specification cannot be met are suspended.

Recurring Query Processing. In recurring systems [10, 11], a recurring query Q is registered, where w and s are defined as configuration parameters. Once registered, the system periodically triggers the execution of Q according to its w and s parameters. Given a recurring query Q, the recurring system [10] pre-processes the input data and subdivides the input files into smaller segments, called panes, with a refined granularity. Each pane is an individual file F_j on HDFS.

An execution E_i processes a sequence of files (F_{x+1} to F_{x+n}), where n is the number of files contained in the window w. When the window of interest w slides by s, any overlapping data files between the two consecutive executions, E_i and E_{i+1}, do not need to be processed again. Therefore, the pane-based partitioning divides a single query execution into a sequence of subtasks over non-overlapping pane inputs, each producing partial results. These partial results are then combined to generate the final desired results. This execution strategy reduces the **unnecessary I/O and CPU costs** otherwise associated with repeated work across overlapping windows [10].

Although the state-of-the-art recurring query system [10] avoids repetitive data processing caused by window and slide semantics, it still strictly processes all recurring query executions to completion in isolation without supporting the preferences specified by the users. In this work, we extended their model to also accept a preference function $pref$ as part of a recurring query.

3. PRO PROBLEM FORMULATION

In this section, we further describe the preference functions specified in the recurring query. Based on these preference functions, we design a metric called *penalty score* (ps) that indicates how preferable each execution is. We then utilize this metric to formulate our optimization goal.

3.1 PRO Preferences

The *penalty score* is defined based on the preference function $\mathcal{U}$ in the PRO query model.

DEFINITION 1 (PENALTY SCORE). *For a query Q and its execution E_i, one preference function $\mathcal{U}$ maps the cost of an execution E_i to a penalty score ps with ps between 0 (maximal preference) and 1 (minimal preference).*

EXAMPLE 1. *A user may be asked to minimize the resource to process a recurring query due to the limited availability of resources of an in-house cluster. In such case, the execution with the lowest cost is the most preferred. This*

preference function can be expressed as:

$$\mathcal{U}(E_i) = 1 - \frac{cost_{min}}{cost(E_i)} \quad (1)$$

where $cost_{min}$ denotes the execution with the lowest estimated processing cost, and $cost(E_i)$ denotes the estimated cost of the execution E_i.

EXAMPLE 2. *A user may favor to update a recommendation model at night (from 8pm to 2am) due to higher user engagements. Thus the executions at night are considered to be the most important, whereas the executions during the day (from 2am - 8pm) are less important. Equation 2 shows such a preference function $\mathcal{U}(\cdot)$ for all E_i (i = 1,...n):*

$$\mathcal{U}(E_i) = \begin{cases} 0.5 & for \quad 8pm < T_i < 2am \\ 1 & for \quad 2am < T_i < 8pm \end{cases} \quad (2)$$

Note that although the executions during the day (from 2am - 8pm) are less important, they can still be processed when the in-house cluster has extra resources.

PRO Hybrid Preference Function. Our preference function $\mathcal{U}$ can also be a combination of multiple preference functions. For ease of elaboration, here we assume that these preference functions are independent. The users thus can specify the relative importance of each function by assigning relative weights. Each user may provide as many preference functions as desired along with the associated weight factors to express their importance. The hybrid penalty score of *one execution* E_i thus becomes:

$$\mathcal{U}(E_i) = \sum_{j=1}^{n} \alpha_j \mathcal{U}_j(E_i) \quad (3)$$

where $\sum_{j=1}^{n} \alpha_j = 1$ and $\alpha_j \geq 0, \forall j \in [1, n]$. For example, having $\alpha_1 = \alpha_2 = 0.5$ means that both preference functions $\mathcal{U}_1(\cdot)$ and $\mathcal{U}_2(\cdot)$ are equally important for E_i to the user.

3.2 Problem Definition

DEFINITION 2 (PROBLEM DEFINITION). *Given a recurring query $Q(w, s, pref)$ with window w, slide s and a preference specification $pref$ (i.e., $pref(m, k, \mathcal{U})$), the **preference-aware recurring query optimization problem** is to find an execution configuration ε with the size $|\varepsilon| = k$, selected from the total number of m executions that minimizes the penalty score of a recurring query Q. That is,*

$$Minimize: \; ps(Q) = \sum_{E_i \in \varepsilon} \mathcal{U}(E_i)$$

where $\mathcal{U}(E_i)$ is the total penalty score of each execution $E_i \in \varepsilon$ as per Equation 3.

Finding the optimal solution for this problem requires enumerating all possible combinations of query executions for Q, which is prohibitively expensive. Consecutive executions of a recurring query may share one or more overlapping input files. This overlap among recurring executions causes the cost of one execution to depend on its preceding executions included in the execution configuration. Any solution to our execution selection problem would need to tackle this *stochastic cost* nature of recurring queries. In the following sections, we will first describe our proposed solutions to the above problem with a fixed value of k. Then, we will discuss a relaxed problem in which the value of k can vary.

4. PRO PROBLEM ANALYSIS

4.1 Modeling Transition Costs

As described in Section 2, an execution E_i processes a sequence of files (F_{x+1} to F_{x+n}), where n is the number of files contained in a window w. To avoid repetitive data loading among consecutive executions, the intermediate results generated from previous executions are cached for subsequent reuse. Hence, the unnecessary I/O costs and the corresponding execution time resulting from overlapping windows for a query can be eliminated.

Due to the caching of intermediate results, the cost of a query execution E_i on a file F_x can differ significantly on whether some preceding executions have already processed F_x. Therefore, the cost model now should account for the caching mechanism as follows.

$$cost(F_x) = \begin{cases} cost_L + cost_P & F_x \; is \; new \\ cost_P & F'_x s \; cache \; is \; available \end{cases} \quad (4)$$

where $cost_L$ and $cost_P$ denote the cost of data loading and computation of an execution E_i, respectively. The *cost of an execution* E_i is thus defined below.

DEFINITION 3 (EXECUTION COST). *Given a recurring query Q, E_i is one query execution of Q over n input files $\mathbb{F}_i = \{F_{x+1}, F_{x+2}, ..., F_{x+n}\}$. The cost of this execution $cost(E_i)$ is defined as:*

$$cost(E_i) = cost(\mathbb{F}_i) = \sum_{l=1}^{n} cost(F_{x+l}) \quad (5)$$

where $cost(F_{x+l})$ is defined in Equation 4.

Given that the consecutive executions are interdependent and the cost of processing one execution may be affected by its preceding executions, we define the notion of a *transition cost* to model this unique characteristic of the recurring query execution.

DEFINITION 4 (TRANSITION COST). *Given two executions E_i and E_j ($1 \leq i < j$) of a recurring query Q, and the file sets $\mathbb{F}_i$ and $\mathbb{F}_j$ processed by E_i and E_j respectively, the costs of processing these two executions independently are $cost(E_i)$ and $cost(E_j)$, respectively. Then the **transition cost** (denoted as $cost(E_{i,j})$) from E_i to E_j is defined as follows:*

$$cost(E_{i,j}) = cost(\mathbb{F}_j \backslash \mathbb{F}_i) \quad (6)$$

where $\mathbb{F}_j \backslash \mathbb{F}_i$ denotes the files associated only with E_j but not with E_i.

Note that this *transition cost* can be utilized to calculate the penalty score ps if a cost-based preference function is provided as part of the $pref$ parameter. For example, assuming execution E_i had previously been chosen to be processed, then the penalty score ps of E_j is $\left(1 - \frac{cost_{min}}{cost(E_{i,j})}\right)$.

4.2 Modeling the PRO Optimization Problem

With this notion of *transition cost*, the uncertainty in the execution cost can be eradicated by modeling our problem of selecting execution configurations with a weighted directed acyclic graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Each vertex in graph $\mathcal{G}$ represents an execution (e.g., $v_i \in \mathcal{V}$ represents execution E_i), and the weight of each directed edge $e_{i,j} \in \mathcal{E}$ from vertex v_i to vertex v_j ($\forall v_i, v_j \in \mathcal{V}$ and $i < j$) represents the penalty score ps (Definition 1) of choosing $E_j \in \mathcal{E}$ right after E_i.

DEFINITION 5. *Given a time period consisting of m executions $\{E_1, E_2, ..., E_m\}$, a directed acyclic graph* ERG *$\mathcal{G} = (\mathcal{V}, \mathcal{E})$, called* ***Execution Relation Graph*** *is defined as*

1. *$v_0 \in \mathcal{V}$ denotes the root node (v_0 leads to the first selected execution),*
2. *$\forall i \in [1, m]$, $v_i \in \mathcal{V}, i \in [1, m]$ represents execution E_i,*
3. *a directed edge $e_{i,j} = (v_i, v_j) \in \mathcal{E}$ exists only if $v_i, v_j \in \mathcal{V}$ and $i < j$. This directed edge denotes the decision of choosing E_j directly after choosing E_i. The weight of $e_{i,j}$ denotes its penalty score (Definition 1), which is always positive.*

DEFINITION 6. *Given a* ERG *graph $\mathcal{G}$ by Definition 5, a* ***directed length-k path*** *$\mathcal{P}_i$ in $\mathcal{G}$ is defined as follows:*

1. *$\mathcal{P}_i$ starts at the root node v_0,*
2. *$\mathcal{P}_i$ is composed of k edges (denoted as $|\mathcal{P}_i| = k$).*

5. PRO OPTIMIZER

To solve the core preference-aware recurring query optimization problem defined in Section 4.2, we have transformed it into a minimum weight length-k path graph problem. We now propose the PRO optimizer consisting of a family of algorithms for finding a length-k path with minimum weight, taking **both** the *length of a path* and the *weight of a path* into consideration. First, we present an exhaustive search-based solution with pruning. Thereafter, we propose a lightweight dynamic programming approach (PRO-OPT) to produce an optimal solution. The PRO-OPT is extensible to execution configurations with varying numbers of executions. Lastly, we design an adaptive methodology to tackle fluctuations in the data rates.

5.1 Enhanced Exhaustive Search

To find minimum weight length-k paths as per Definition 6 in an ERG graph $\mathcal{G}$, the PRO optimizer needs to enumerate all possible length-k paths and to select the one with the minimum weight. Unfortunately, the search space can be shown to be exponentially huge [7]. There is not one unique solution, i.e., there can be more than one length-k path with the identical minimum weight.

A basic search algorithm would approach this problem by traversing the graph $\mathcal{G}$ in a breadth-first search manner from root v_0. In the i-th iteration, it forms all possible length-i paths. After k iterations, it produces all length-k solutions. It then selects the one with minimal weight. If more than one length-k path with equal minimal weight exist, then this approach randomly selects one of them to be the final execution configuration. The exhaustive search solution is exponentially expensive as there are $\binom{m}{k}$ length-k paths in an ERG $\mathcal{G}$.

While the search space is intractable, one key observation is that certain subset of vertices in $\mathcal{G}$ do not need to be considered any further as part of the final solution at each iteration without losing optimality (see Lemma 1).

LEMMA 1 (VERTEX PRUNING). *Given an* ERG *$\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a desired path length k, then the out degree of $v_a \in \mathcal{V}$ is denoted as $deg^+(v_a)$. In order to be considered fo r a length-k path at the* ***i-th iteration****, the vertex v_a must satisfy:*

$$deg^+(v_a) \geq k - i. \tag{7}$$

5.2 Dynamic Programming-Based Approach

We now propose a dynamic programming-based PRO-OPT approach that reduces search complexity. The key idea of PRO-OPT is to exploit tabulation by building a series of reusable *sub-paths* to form a length-k solution. An important result is that PRO-OPT guarantees to produce an **optimal** solution with a *pseudo-polynomial* time complexity.

DEFINITION 7 (MINIMUM WEIGHT SUB-PATHS). *Let $\mathcal{P}_i^{(h)}$ denote the set of all length-h paths starting at vertex v_i and ending at any other vertex in a* ERG *$\mathcal{G}$. Then we define $p_{i,min}^{(h)}$ as the minimum weight length-h path in $\mathcal{P}_i^{(h)}$.*

Next we show that we can form a minimum weight length-h path by adding one new edge to the current minimum weight sub-path of length $h-1$.

DEFINITION 8 (EXTENSION OF PATHS). *Given an* ERG *$\mathcal{G} = (\mathcal{V}, \mathcal{E})$ by Definition 5 and an integer k as the desired path length. All sub-paths with length h less than k can be extended according to the following equation:*

$$w(p_{i,min}^{(h)}) = \begin{cases} min\{w(e_{i,j})\}, & h = 1 \\ min\{w(e_{i,j}) + w(p_{j,min}^{(h-1)})\}, & h \in [2:k] \end{cases} \tag{8}$$

where $i \in [0, m]$, $j \in [i+1, m-h+1]$. $w(e_{i,j})$ and $w(p_{j,min}^{(h-1)})$ represent the weight of edge $e_{i,j}$ and the minimum weight of length-$(h-1)$ sub-path starting from v_j, respectively.

For a given vertex v_i, only edges starting at v_i and of length less than or equal to $m-h$ (i.e., $e_{i,i+1}$, $e_{i,i+2}$,...,$e_{i,m-h}$) are considered to form the minimum weight length-h sub-path $p_{i,min}^{(h)}$. Multiple sub-paths starting from one vertex can have the same minimum weight. Hence, all minimum weight sub-paths $p_{i,min}^{(h)}$ from one vertex and their respective associated minimum weight are stored in the Dynamic Search Table (DST) to assure re-computations are avoided.

PRO-OPT Pruning Strategy. Similar to Lemma 1, not all minimum weight sub-paths need to be considered. Instead, some sub-paths can never form a length-k path. Therefore, we now reduce the Dynamic Search Table (DST) by pruning these useless sub-paths.

Our formal PRO-OPT algorithm given in Algorithm 1 exploits both the sub-path extension method (Definition 8) and dynamic pruning strategy to produce a minimum weight length-k path.

The proposed dynamic pruning strategy does not affect the correctness of PRO-OPT as it prunes sub-paths if and only if they cannot be used to produce any length-k paths. We now state the correctness of PRO-OPT in Lemma 2.

LEMMA 2 (OPTIMALITY OF PRO-OPT). *Given an* ERG *$\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the length-$k$ path produced by PRO-OPT has the minimal weight among all possible length-k paths.*

5.3 Discussion

Problem Relaxation. As mentioned in Section 3.2, the expected number of executions k can vary due to different reasons. For example, domain experts may want to increase k due to the change of their applications. The number of executions k can also decrease when resources become less available. Thus, PRO optimizer should have the ability to generate a *new* execution configuration ε' with a different

Algorithm 1 PRO-OPT Algorithm

INPUT: Directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and integer k
OUTPUT: A minimum weight simple k-path within G

for $h = 1, h \leq k, h \leftarrow h + 1$ **do**
  for $i = k - h + 1, i < |\mathcal{V}| - k, i \leftarrow i + 1$ **do**
    for $l = i + 1, l < |\mathcal{V}| - 2, l \leftarrow l + 1$ **do**
      if $h = 1$ **then**
        $w(p^h_{i,min}) = min\{w(e_{i,i+l})\}$
        store $w(p^h_{i,min})$ and $p^h_{i,min}$
      else if $2 \leq h < k$ **then**
        $w(p^h_{i,min}) = min\{w(e_{i,i+l}) + w(p^{(h-1)}_{i+l,min})\}$
        store $w(p^h_{i,min})$ and $p^h_{i,min}$
      else if $h = k$ **then**
        $w(p^h_{0,min}) = min\{w(e_{0,l}) + w(p^{(h-1)}_{l,min})\}$
        store $w(p^h_{0,min})$ and $p^h_{0,min}$
        break
      end if
    end for
  end for
end for

number of executions based on the *old* one as needed. Fortunately, our PRO-OPT is designed to build longer sub-paths incrementally based on existing sub-paths stored in DST. Therefore, a smaller size execution configuration ε' is existed in DST. In addition, the minimum weight length-$k + i$ path can be extended from existing length-$k + i - 1$ paths with a linear time complexity. Thereafter, if more than k executions are feasible for the new time period based on the cost estimation (Equation 1) and the available resources, our PRO optimizer will produce an optimal execution configuration (i.e., the minimum weight length-$k+i$ path) for the new time period.

Overlapping m Executions. Another issue concerns that consecutive time periods of m executions may overlap with each other. For example, if we have two consecutive overlapping time periods B_i and B_j $(i < j)$ of m executions each, for any execution $E_i \notin \mathbb{E}_i \bigcap \mathbb{E}_j$, the execution cost of E_i does not need to be changed. However, the costs of all executions that belong to both periods (i.e., $\mathbb{E}_i \bigcap \mathbb{E}_j$) need to be adjusted to account for the fact that repetitive computations are eliminated in the PRO system. Consequently, the edge weights that represent penalty scores in an ERG graph are updated as well. Thereafter, we can simply apply the above discussed adaptive technique to determine whether a re-optimization is needed for the new time period.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup & Methodology

Experimental Infrastructure. We conduct all experiments on a shared-nothing cluster with one master node and 39 slave nodes.* Each node has a configuration of 16 core AMD 3.0GHz processors, 32GB RAM, 250GB disk, and these 40 nodes are interconnected with 1Gbps ethernet. Each node runs CentOS of version 6.5, Java 1.7 and Hadoop 1.2.1. Each node is configured to concurrently run up to 8 map and 8 reduce tasks. The sort buffer size is set to 512MB. We keep the default replication factor to 3.

*This cluster is supported by NSF Grant CNS-1305258.

Datasets and Queries. We use two real world datasets for our experiments. The Wikipedia page editing log (400GB), and the current version of all article pages from Wikipedia (54GB) [1] being modified and updated continuously. We focus on operations that are critical for emerging data analytical tasks, such as aggregation queries and top-k queries.

Algorithms and Systems. We implemented our PRO optimizer with the proposed algorithms, PRO-EES and PRO-OPT on top of a distributed platform, in this case Apache Hadoop [3]. In addition, we implemented a depth first search algorithm PRO-DFS for the purpose of comparison. Native Redoop [10] is used in runtime performance experiments. Since there is no support for preference-aware optimization in native Redoop, we also configure a variation of Redoop as k-Redoop, to only process first k executions in each time period to make a fair competitor in robustness experiments.

Metrics & Methodology. We evaluate *PRO runtime* performance by varying the input data volume and the number of machines in the cluster and we measure the execution time-a common metric for data management systems [11]. We evaluate the *result quality* using ranking quality measurement methods, namely Kendall's tau [9], Normalized Discounted Cumulative Gain (NDCG) [8] and Expected Reciprocal Rank [4]. Each method produces a precision result pr and we define our error rate er to be $er = 1 - pr$. Thereafter, we compare error rates of PRO-OPT with native Redoop (i.e., all executions are being executed). We utilize the preference function $\mathcal{U}_1$ in Table 1 for the above experiments. We verify the robustness of PRO optimizer with various preference functions in optimizer robustness experiments.

#	Preference Functions
$\mathcal{U}_1(E_i)$	$1 - \frac{cost_{min}}{cost(E_i)}$
$\mathcal{U}_2(E_i)$	$\begin{cases} 0.5 & E_i.priority = high \\ 1 & E_i.priority = low \end{cases}$
$\mathcal{U}_3(E_i)$	$\sum_{j=1}^{n} \alpha_j \mathcal{U}_j(E_i)$

Table 1: Preference Functions Used in the Experiments

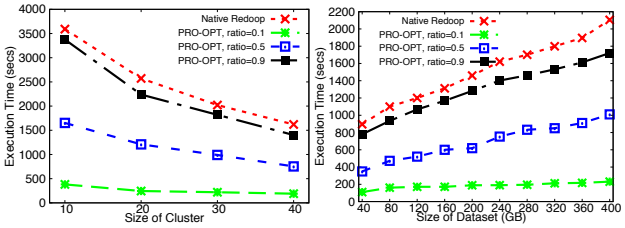
6.2 PRO Runtime Performance

Varying the cluster size. Figure 1(a) demonstrates the scalability of PRO by varying the number of nodes in the computing cluster. Compared to native Redoop, PRO-OPT shows an average of 323% savings in processing time from cluster size of 10 to 40 and up to 840% savings in processing time when the number of nodes in the cluster is 10 and execution selection ratio is 0.1.

Varying the data size. Figure 1(b) illustrates the execution times of PRO-OPT solution with different data sizes ranging from 40GB to 400GB per a time period of m executions. PRO-OPT resource utilization significantly drops compared to processing all executions. For example, PRO-OPT saves up to 9 fold in execution time when the data size is 400GB compared to native Redoop processing all executions. The execution time of PRO grows much slower than native Redoop system for increased data sizes. For instance, we observe the increasing rate of execution time from native Redoop is 35% higher than PRO. The reason is that since we use resource-driven preference function, PRO-OPT always selects executions that will use the least resources.

6.3 Result Quality

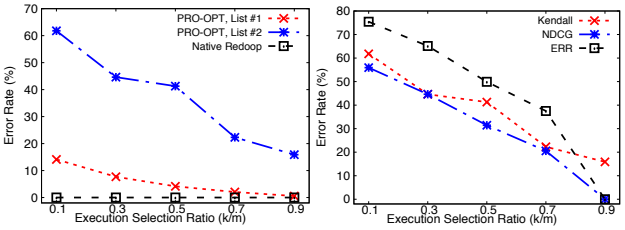
We use PRO-OPT with execution selection ratios ranging from [0.1, 0.9] to compute two lists of frequent items, i.e.,



(a) Vary Cluster Size (b) Vary Quality Method

Figure 1: PRO Runtime Performance

top-n frequent words (List #1) and the most frequent editors in Wikipedia articles (List #2). We then compare error rates between the results from PRO-OPT and Redoop. In Figure

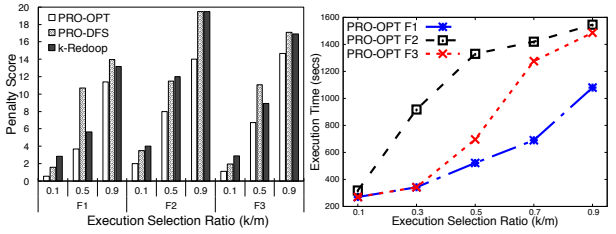


(a) Vary Item List (b) Vary Quality Method

Figure 2: PRO Error Rate

2(a), we notice the error rate is much higher for List #2 than List #1. For List #1, the error rate ranges from 14% to 0.5% when execution selection ratio is between 0.1 and 0.9, respectively. For List #2, the error rate is 61.8% when there is 10% of all executions within one time period are processed and 15.8% when 90% of all executions within one time period are processed. The reason is the ranking of frequent words is more stable than the list of most frequent editors. In the best case, we have an error rate of 0.06% when the execution selection ratio is 0.9 with NDCG in Figure 2(b). In the worst case, when execution selection ratio is 0.1 with ERR method, we have an error rate of 75%. We observe that the error rate of all three ranking measurement methods show the same significantly decreasing trend as the execution selection ratio increasing from 0.1 to 0.9 as a result of processing more input data. The trade-off is that PRO uses far less resources compared to native Redoop, yet still is able to produce high quality results.

6.4 Robustness of PRO Optimizer



(a) Penalty Score (b) Execution Time

Figure 3: Varying Preference Functions

Next, we examine the effect of varying preference functions from Table 1 on the penalty scores and execution times. For that, we vary the execution selection ratio from [0.1, 0.9] on the x-axis. The number of executions in each time period is set to 30 and input data size is 240GB for one time period. We randomly select 15 out of 30 executions as "low priority" (penalty score is 1). The rest are set to "high priority" (penalty score is 0.5) in preference function $\mathcal{U}_2$. We combine functions $\mathcal{U}_1$ and $\mathcal{U}_2$ using function $\mathcal{U}_3$ with relative importance of each function ($\mathcal{U}_1$ and $\mathcal{U}_2$) equally to 0.5. Figure 3(a) shows that PRO-OPT always produces execution configurations with minimum penalty scores compared to PRO-DFS and k-Redoop for all three preference functions. Figure 3(b) shows that the processing times of execution configurations generated by PRO-OPT with different preference functions can differ drastically. For example, when execution selection ratio is 0.5, the execution configuration produced by PRO-OPT with the function $\mathcal{U}_2$ takes 61% longer time to process than PRO-OPT with the function $\mathcal{U}_1$ and 47% longer compared to PRO-OPT with function $\mathcal{U}_3$. The reason is the execution configurations actually have different sets of executions according to different preferences.

7. CONCLUSION

This paper presents the first preference-aware recurring query processing solution. Our PRO system offers 3 key innovations. (1) The preference-aware recurring query model established for PRO integrates recurring query processing with various preferences specified by application developers. (2) The proposed execution relation graph models the interdependencies and stochastic execution costs among executions. (3) Our novel PRO-OPT algorithm effectively produces an optimal execution configuration with a pseudo-polynomial time complexity. Its efficiency enables PRO to adaptively re-optimize when faced with data fluctuations. Lastly, our experimental results demonstrate that our PRO solution consistently outperforms state-of-the-art technologies by 9 fold in a large range of scenarios.

8. REFERENCES

[1] Wikimedia downloads. https://dumps.wikimedia.org.

[2] S. Agarwal, B. Mozafari, et al. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[3] Apache. Hadoop. http://hadoop.apache.org.

[4] O. Chapelle, D. Metlzer, Y. Zhang, and P. Grinspan. Expected reciprocal rank for graded relevance. In *CIKM*, pages 621–630, 2009.

[5] J. Gama, R. Sebastião, et al. Issues in evaluation of stream learning algorithms. In *SIGKDD*, pages 329–338, 2009.

[6] Í. Goiri, R. Bianchini, S. Nagarakatte, et al. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ASPLOS*, pages 383–397, 2015.

[7] A. Hassidim, O. Keller, et al. Finding the minimum-weight k-path. In *Algorithms and Data Structures*, pages 390–401. Springer, 2013.

[8] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM TODS*, 20(4):422–446, 2002.

[9] M. G. Kendall. A new measure of rank correlation. *Biometrika*, pages 81–93, 1938.

[10] C. Lei, E. A. Rundensteiner, and M. Y. Eltabakh. Redoop: Supporting recurring queries in hadoop. In *EDBT*, pages 25–36, 2014.

[11] C. Lei, Z. Zhuang, E. A. Rundensteiner, and M. Eltabakh. Shared execution of recurring workloads in mapreduce. *PVLDB*, 8(7):714–725, 2015.

[12] D. Logothetis, C. Trezzo, K. C. Webb, et al. In-situ mapreduce for log processing. In *USENIXATC*, pages 9–9, 2011.

[13] H. Ryser. Combinatorial mathematics. *The Carus Mathematical Monographs, No. 14*, 1963.

[14] J. Sarma. Hadoop at facebook. https://www.facebook.com/notes/facebook-engineering/hadoop/16121578919.

[15] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *SIGMOD*, pages 1125–1134, 2013.

[16] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-ola: Generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*, pages 913–918, 2015.