Online Query Scheduling on Source Permutation for Big Data Integration

Zimu Yuan, Shusheng Guo
University of Chinese Academy of Sciences, China
Institute of Computing Technology, Chinese Academy of Sciences, China
{yuanzimu, guoshusheng}@ict.ac.cn

Abstract—Big data integration could involve a large number of sources with unpredictable redundancy information between them. The approach of building a central warehousing to integrate big data from all sources then becomes infeasible because of so large number of sources and continuous updates happening. A practical approach is to apply online query scheduling that inquires data from sources at runtime upon receiving a query. In this paper, we address the Time-Cost Minimization Problem for online query scheduling, and tackle the challenges of source permutation and statistics estimation to minimize the time cost of retrieving answers for the real-time receiving query. We propose the online scheduling strategy that enables the improvement of statistics, the construction of source permutation and the execution of query working in parallel. Experimental results show high efficiency and scalability of our scheduling strategy.

I. INTRODUCTION

Big data not only means a large volume of data, but also indicates variety that data can be from a large number of sources. The integration of big data that unifies varieties of data from widely-distributed sources can act as the foundation of information conformity for applications. For this big data integration, it has two major changes compared with traditional data integration. First, continuous updates may exist in a fair amount of data sources. As a consequence, the traditional way of building a central warehousing to integrate data from sources would become infeasible. A practical way is to apply online scheduling that inquires data from sources at runtime upon receiving a query. Second, a large proportion of unpredictable redundant data exist between sources. Thus, the query results returned from different sources should be judged, removing the repetitive ones. To deal with these two challenges, an intelligent online query scheduling could be designed to answer the query with non-redundant results in the least possible time.

In this paper, we set out to address this online query scheduling problem. In its scenario, as illustrated in Figure 1, the querier that does not cache any data itself and the domains that act as data sources are independent each other. For query scheduling, the querier may firstly arrange the domains in a permutation. Then, upon receiving a query, the querier sequentially inquires each domain (or few domains in parallel) at a time following the permutation. When obtaining new results from a domain, the querier compares them with previous results received from other domains, and removes the repetitive ones. Finally, the querier returns the results after receiving enough ones. We also found that the scenario depicted in Figure 1 is common in research and practical application. In application, this style of online query on multiple sources is usually seen in Aggregator. For example, Google News, a news aggregator, watches updates from more than 4500 worldwide news sources, and exhibits non-redundant news to the readers; KAYAK.com, a trip aggregator, shows similar trips obtained from hundreds of travel sites. More aggregator applications can be referred in [1]. Besides, in domain-centric research, related studies mainly focus on integration of the knowledge about a topic from widespread sources. As shown in the experiment of [3], about 5000 sources are needed to be queried on for acquiring 95% knowledge of a topic due to a large portion of redundant knowledge. Also, similar results in other topics are given in [3].

For reduction of query time, in our experience of online scheduling, we have found that the critical point lies in the permutation of sources. Each data source is with a different access time, a different transfer time, and a different proportion of intersection data (namely redundant information) with other sources. Choosing a good permutation of sources can reduce the total time cost for a query, especially with much more significantly reduction for the query in a long run on a large number of sources. To better describe the permutation choice, we give a simple example on query $Q_k$ (namely a query $Q$ for acquiring $k$ tuples) here:

Example 1. Consider three sources $S_1$, $S_2$ and $S_3$ shown in Figure 2. For simplicity, set the access time $ta_1 = ta_2 = ta_3 = 0$ and the transfer time of retrieving a tuple $tr_1 = 0.7 ms$, $tr_2 = 1.1 ms$ and $tr_3 = 1.5 ms$ for $S_1$, $S_2$ and $S_3$.
respectively. Upon receiving a query $Q_k$, the querier starts to estimate the statistics of sources, and has the estimated value for data items, and thus data integration system could apply the pruning techniques that only probes the critical statistics on-the-fly.

- We propose the online scheduling strategy that enables the improvement of statistics collection, the permutation construction of OnlinePerm algorithm and the execution of query on sources working in parallel, so as to reduce the total time cost for the query.
- We conduct experiments on physical sources. The experiment results show that our scheduling strategy is scalable and can significantly reduce the time cost.

The rest of this paper is organized as follows. Section II presents the related work. Section III gives the overview of online query scheduling. Section IV presents the OnlinePerm algorithm for source permutation. Section V proposes the mechanism of two-stage detection for statistics collection. Section VI describes the scheduling strategy. Section VII presents the experiment results. Section VIII concludes this paper.

II. RELATED WORK

There exist some research work on consideration of intersection or redundancy information for source selection. In [5], sources were classified into domains (e.g. Journal paper, Conference paper), and probabilistic intersections between domains were considered to select sources with more answers. In [12], the challenges on calculating the exponential number of source intersections were discussed to provide useful knowledge for source permutation. In [7], the StatMiner system was proposed to learn the union and intersection statistics between classes (including sources and queries) and provide static selection of sources with more relevant answers for each class. In [2], the maximize number of duplicates were found between sources to answer the Join queries for entities. In [9], the copying relationships between sources known in advance were considered to determine the permutation of sources with the target of cost-minimization and maximum-coverage. In [4], possible intersections or duplicates between sources are studied to estimate the expected value for data items, and thus data integration system could construct a permutation of sources by the accuracy of data items. In [11], the answers for a query were estimated by the statistics (including intersection knowledge) obtained by other queries for the crowdsourced system, and thus a permutation
of sources can be given by sorting the sources with the number of answer tuples in descending order. In [9], the OASIS system was proposed to online collect intersection statistics and find a permutation of sources to maximize the Area-under-the-curve value for retrieving all answer tuples from sources.

**Summary of differences:** Our work differentiates with related literatures in following aspects: (1) we take the real-time receiving query $Q$ of $k$ tuples as input, (2) we focus on the Time-Cost Minimization problem for source permutation, (3) we propose a two-stage detection mechanism that combines the offline and online collection processes to estimate intersection statistics between sources, and (4) we present the online scheduling strategy that enables source permutation, statistics collection and query execution working in parallel.

**III. OVERVIEW**

**A. Model and Problem**

Given a set of sources $Y = \{S_1, S_2, ..., S_l\}$ and a permutation $\Pi(\hat{Y})$ for a query $Q$, the query rate $v_i$ of a source $S_i \in Y$ can be written as follow:

$$v_i = \frac{|S_i| - |\cap S_i|}{ta_i + tr_i|S_i|}$$

(2)

In (2), $|S_i|$ is the total number of result tuples for $Q$ in source $S_i$; $|\cap S_i|$ is the intersection tuples that has been transferred to the querier from other sources (that is, these intersection tuples are in $|S_j \cap S_i|$, where $S_j$ is prior to $S_i$ in $\Pi(\hat{Y})$); $ta_i$ and $tr_i$ denote the access time and the transfer time of a tuple from $S_i$ to the querier respectively.

Suppose that the permutation $\Pi(\hat{Y})$ has $\Pi(\hat{Y}) = S_{(1)}S_{(2)}...S_{(k)}$, $\sum_{i=1}^{k-1}(|S_i| - |\cap S_i|) < k \leq \sum_{i=1}^{k}(|S_i| - |\cap S_i|)$, and only one query thread is running in the querier, we have the average rate $v_{avg} = \frac{\sum_{i=1}^{k}(|S_{(i)}| - |\cap S_{(i)}|)}{\sum_{i=1}^{k}(ta_{(i)} + tr_{(i)}|S_{(i)}|)}$ and the time cost $T(Q_k(\Pi(\hat{Y}))) = \frac{k}{v_{avg}}$. The optimal permutation $\Pi_{opt}(\hat{Y})$ of which are the union of tuples for query $Q_k$ from $S_i$, $i = 1, 2, ..., l$, and the other $|\hat{Y}|$ of which are from the newly generated tuples for query $Q_k$. Create $l$ subsets $S’_{(1)}, S’_{(2)}, ..., S’_{(l)}$ of $\hat{Y}$. Each subset $S’_{(i)}$ contains $|S_i|+|\hat{Y}|$ elements that are the tuples for query $Q_k$ from source $S_i$ and the newly generated ones.

For a query $Q$ of $|\hat{Y}|$ and $|\hat{Y}|^2$ tuples, if there exist an optimal permutation $\Pi_{opt}(\hat{Y})$ of $m$ sources, we can easily see that the union of all elements in these sources cover all the elements of the universal set. Thus, the reduction from TMP to Set Cover Problem is established. In turn, suppose that there exist smallest numbers of $m$ subsets $S’_{(1)}$, $S’_{(2)}$, ..., $S’_{(m)}$ whose union equals the universal set $\hat{Y}$. We firstly construct a permutation $\Pi(\hat{Y})$ by randomly arranging these $m$ subsets, and set $ta_i = 0 ms$ and $tr_i = 1 ms$ for all $S_i \in \hat{Y}$. Then, we have the time cost for query $Q$ of $|\hat{Y}|+|\hat{Y}|^2$ tuples:

$$T(Q_{|\hat{Y}|+|\hat{Y}|^2}(\Pi(\hat{Y}))) \leq m(|\hat{Y}|+|\hat{Y}|^2)$$

(3)

Consider any other permutation $\Pi(Y)$ with $m+1$ subsets:

$$T(Q_{|\hat{Y}|+|\hat{Y}|^2}(\Pi(Y))) \geq (m+1)|\hat{Y}|^2$$

(4)

For this instance of TMP, we have $T(Q_{|\hat{Y}|+|\hat{Y}|^2}(\Pi(Y))) \leq T(Q_{|\hat{Y}|+|\hat{Y}|^2}(\Pi(Y)))$. Hence, the optimal permutation $\Pi_{opt}(\hat{Y})$ with $m$ subsets exists if the union of these $m$ subsets covers the universal set.
The decision version of TMP is to decide the query time cost for a given permutation $\Pi(Y)$. The total time cost for $\Pi(Y)$ can be calculated out in $O(l)$ time. Therefore, TMP is NP-complete.

Since TMP is NP-complete, the basic solution of traversing all permutation of sources will soon become unmanageable when the number of sources increase. We consider an alternative scalable algorithm to construct the permutation for TMP. Naturally, to incrementally construct the permutation for TMP when the number of sources increase. We consider an alternative scalable algorithm to construct the permutation for TMP. Naturally, to incrementally construct the permutation $\Pi(Y)$, a greedy algorithm can be planned to sequentially choose a source $S_i$ that has the fastest query rate $v_i = \frac{|S_i| - |\cap S_i|}{t_{a_i} + t_{r_i} + t_{s_i}}$ at each iteration. For the example of Figure 4, source $S_1$ is first selected since its query rate $v_1 = 1.43 \text{ tuple/ms}$ is the fastest among these three sources. Afterward, source $S_2$ is then extracted with $v_2 = \frac{125 - 35}{104 \times 1} = 0.65 \text{ tuple/ms}$. Finally, source $S_3$ is chosen with $v_3 = \frac{53 - 50}{104 \times 1.5} = 0.53 \text{ tuple/ms}$. Another approach is to examine the residual tuples of each source and choose the source with maximal residual tuples at each iteration. For the example of Figure 5, source $S_2$ is first chosen since it has the maximal residual tuples $|S_2| = 125$. Then, source $S_2$ and $S_1$ are selected sequentially with residual tuples of 65 and 10 respectively. However, as shown in Example 4, neither the permutation $S_1S_2S_3$ by greedily considering fastest query rate nor $S_2S_3S_1$ by greedily considering maximal residual tuples is an optimal solution.

Actually, we investigate both the effect of query rate and residual tuples on source permutation for TMP. As can be seen in Figure 4, curve $S_1S_2S_3$ returns more tuples compared to curve $S_2S_1S_3$ with the same time before the crosspoint of $(96.8, 106.4)$, and the result reverses after the crosspoint. For a given query $Q_k$, we could find a better permutation by discovering possible crosspoints. Next, we give two observations that reveal the correlation of query rate and residual tuples on crosspoint appearance.

**Observation 1.** The crosspoint of two permutation curves appears when a swap happens between a source with faster query rate and another source with more residual tuples.

In Figure 5(a), source $S_1$ has faster query rate and less residual tuples compared to source $S_2$ at time $t_0$. The query rate of the curve $\Pi_1(Y)$ slows down after $S_1$ has been queried while the query rate of the curve $\Pi_2(Y)$ keeps and catches up $\Pi_1(Y)$ at that time since $S_2$ has more tuples than $S_1$. The crosspoint of curve $\Pi_1(Y)$ and $\Pi_2(Y)$ exists when $S_2$ has enough more tuples and a little less query rate than $S_1$.

**Observation 2.** The crosspoint of two permutation curves does not exist when a swap happens between two sources that have few intersection tuples.

For the example in Figure 5(b), the query rate $v_2$ of $S_2$ has $v_2(\Pi_2(Y)) = v_2(\Pi_1(Y)) = v_2(\Pi_2(Y)) = v_2(\Pi_1(Y)) = v_2(\Pi_2(Y))$ for the curve $\Pi_1(Y)$ and $\Pi_2(Y)$. If source $S_1$ has fewer intersection tuples with $S_2$, that is, $|S_1 \cap S_2|$ is a small value, we have $v_2(\Pi_2(Y)) \approx v_2(\Pi_2(Y))$ and $v_1(\Pi_1(Y)) \approx v_1(\Pi_2(Y))$ for $S_1$. In this case, the crosspoint of these two curves does not exist.

The Observation 1 and 2 indicate heuristics for designing the Online Permutation (OnlinePerm) Algorithm. Before describing it, we firstly present the inputs, functions and algorithms on which the OnlinePerm algorithm is built. Given a query $Q$ of $k$ tuples, the following inputs provide essential information for the construction of source permutation:

- **Statistics Input $SI$:** The statistics are from the set $Y = \{S_1, S_2, \ldots, S_i\}$ of sources, including the access time cost $t_{a_i}$, time cost $t_{r_i}$ of querying a tuple, the number of tuples $|S_i|$ answering the given query $Q$ and the number of intersection tuples $|\cap S_i|$ for any permutation of sources, $i = 1, 2, \ldots, l$. Let $|\cap Y|$ to denote $|S_i|$ and $|\cap S_i|$ for $S_i \in Y$. $|\cap Y|$ is unknown or imprecise upon receiving $Q$ at runtime. In Section IV, we will show how to estimate $|\cap Y|$ by two-stage detection.
- **Intersection Threshold $\theta_{sp}$:** For a source $S_i$, its intersection proportion with another source $S_j$ can be computed by $\frac{|S_i \cap S_j|}{|S_j|}$. If the value of $\frac{|S_i \cap S_j|}{|S_j|}$ is below the threshold $\theta_{sp}$, we can ignore the source $S_j$ for $S_i$ in the permutation construction.

We define the following functions:

- **$InSec$:** Given a permutation $\Pi(Y)$ of selected sources for the query $Q_k$ and a set $Y_e$ of unselected sources, function $InSec$ returns the number of intersection tuples $|\cap S_i|$ with the sources in $\Pi(Y)$ for $S_i \in Y_e$.
- **Counter:** Given a permutation $\Pi(Y)$ of selected sources for the query $Q_k$, function $Counter$ counts the total number of tuples $n_{sum} = \sum_{S_i \in \Pi(Y)} (|S_i| - |\cap S_i|)$.
- **Perm2Set:** Supposing that $\sum_{i=1}^{k-1} (|S_i| - |\cap S_i|) < k \leq \sum_{i=1}^{k} (|S_i| - |\cap S_i|)$, the function $Perm2Set$ removes the $i_k$th, $(i_k + 2)$th, ..., $|\Pi(Y)|$th sources from $\Pi(Y)$, and add these sources to the set $Y_e$ of unselected ones.
Algorithm 1: Greedy on Query Rate (GreedyQR)

Input: A query $Q_k$; A permutation $\Pi(Y)$ of selected sources; A set $Y_e$ of unselected sources; Statistics Input SI

Output: The new permutation $\Pi(Y)$ and set $Y_e$; The total number of tuples $n_{sum}$ of $\Pi(Y)$; The average query rate $v_{avg}$ of $\Pi(Y)$

1. $[n_{sum}] = \text{Counter}(\Pi(Y),SI)$;
2. if $n_{sum} \geq k$ then
   3. $[n_{sum},v_{avg}] = \text{Perm2Set}(\Pi(Y),Y_e,SI,k)$;
else
   5. Set a empty source $S_q$ with no tuple;
6. while $n_{sum} < k$ do
5. Set $v_{max} = 0$;
7. $\cap\{S_i,S_j \in Y_e\} = \text{InSec}(\Pi(Y),Y_e,SI)$;
8. foreach Data Source $S_i \in Y_e$ do
9. Calculate the query rate $v_i = \frac{|S_i|-|\cap\{S_i\}|}{ta_i+tr_i|S_i|}$;
10. if $v_i > v_{max}$ then
11. Set $v_{max} = v_i$ and $S_q = S_i$;
end
end
14. $[n_{sum},v_{avg}] = \text{Set2Perm}(S_q,\Pi(Y),Y_e,SI)$;
end
16. return $\Pi(Y),Y_e,n_{sum}$ and $v_{avg}$;

Then, $\text{Perm2Set}$ returns the total number of tuples $n_{sum}$ and the average query rate $v_{avg}$ of $\Pi(Y)$;

- $\text{Set2Perm}$: Given the permutation $\Pi(Y)$ of selected sources and a set $Y_e$ of unselected sources, the function $\text{Set2Perm}$ adds a source $S_q \in Y_e$ to the tail of the $\Pi(Y)$, removes $S_q$ from $Y_e$, and returns the total number of tuples $n_{sum}$ and the average query rate $v_{avg}$ of $\Pi(Y)$;

- $\text{Swap}$: Given $S_i \in \Pi(Y)$ and $S_j \in Y_e$, the function $\text{Swap}$ replaces $S_i$ with $S_j$ in $\Pi(Y)$, removes the sources ranked behind $S_i$ from $\Pi(Y)$, and adds $S_j$ and those removed sources to $Y_e$. Then, $\text{Swap}$ returns the new $\Pi(Y)$ and $Y_e$;

- $\text{Sort}$: Given $S_i \in \Pi(Y)$ and an expression $\text{Exp}$, the function $\text{Sort}$ sorts the sources in $Y_e$ by the value of $\text{Exp}$, discards the sources with the value below the threshold $\theta_{sp}$. Then, $\text{Sort}$ returns the sorted permutation $\Pi(Y)$ for $Y_e$;

- $\text{LockWrite}$: The function $\text{LockWrite}$ writes $\Pi(Y)$ to the shared file for the query $Q_k$ with the update lock;

The Algorithm [1] Greedy on Query Rate (GreedyQR), invokes the function $\text{Counter}$ to check the total number of tuples $n_{sum}$ in $\Pi(Y)$. If $n_{sum} \geq k$, GreedyQR moves extra sources from $\Pi(Y)$ to $Y_e$ by $\text{Perm2Set}$; else greedily selects the source with the maximal query rate and add it to the tail of $\Pi(Y)$ by $\text{Set2Perm}$ until $n_{sum} \geq k$.

The Algorithm [2] Re-Permutations (RePerm), explores the possible swaps between a selected source $S_i \in \Pi(Y)$ and these sources in $Y_e$. Based on the Observation [2] RePerm firstly selects the sources from $Y_e$ by sorting with the value of $\frac{|S_i|}{|\cap\{S_i\}|}$ in descending order, and ignores the source $S_j$ if $\frac{|S_i|}{|\cap\{S_i\}|} < \theta_{sp}, \forall S_j \in Y_e$. Then, RePerm only chooses the source $S_j$ having $|S_i| < |S_j|$ to swap $S_i$ by the knowledge from Observation [1]. After the swap process, RePerm re-ranks $S_i$, the sources that are originally ranked behind $S_i$ in $\Pi(Y)$, and the sources in $Y_e$ by GreedyQR to construct a candidate permutation $\Pi'(Y)$. Finally, RePerm returns the permutation $\Pi'(Y)$ with the maximal average query rate $v_{avg}$ from all candidate permutations.

The OnlinePerm algorithm (Algorithm [3]) is proposed based on the effect of both the query rate and the residual tuples. If the given permutation $\Pi(Y)$ is empty, the OnlinePerm algorithm firstly construct the permutation $\Pi'(Y)$ by GreedyQR that greedily adds the source with the maximal query rate from $Y_e$ sequentially, and invokes $\text{LockWrite}$ to write $\Pi'(Y)$ to the shared memory for query $Q$. (The process of the Query Execution component that works in parallel with the OnlinePerm algorithm use $\text{LockRead}$ to read the permutation $\Pi'(Y)$, and queries on sources sequentially following $\Pi'(Y)$). Since the sources that are more closely to the head of $\Pi(Y)$ are queried firstly, OnlinePerm checks and swaps the sources in $\Pi(Y)$ sequentially by RePerm from its head to tail. As presented above, RePerm tries to find a new permutation $\Pi'(Y)$ based on the knowledge of the correlation between query rate and residual tuples provided by Observation [1] and [2]. Once a new permutation $\Pi'(Y)$ having the average query rate $v_{avg} > v_{avg}$ is found, OnlinePerm invokes $\text{LockWrite}$ to write $\Pi'(Y)$ to the shared memory.

Construct a permutation $\Pi'(Y)$ that arranges sources by query rate $v_{avg} = \frac{|S_{u(i)}|}{i(\sum_{i=1}^{n}|S_{u(i)}|)}$ in descending order.
GreedyQR has the monotonicity property that $\sum_{i=1}^{u(i)} |S_u(i)| \geq k$ and $\sum_{i=1}^{u(i)-1} |S_u(i)| < k$. We have the following theorem for $Q_k$:

**Theorem 2.** The time cost $T(Q_k(\Pi(Y)))$ of $\Pi(Y)$ constructed by OnlinePerm is 
$$\frac{k \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)}{|Y| \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)}$$
approximate to the optimal time cost $T(Q_k(\Pi_{opt}(Y)))$ for TMP.

**Proof:** The time cost $T(Q_k(\Pi_{opt}(Y)))$ is greater than $T(Q_k(\Pi(Y)))$ that ignores the intersection tuples:

$$T(Q_k(\Pi(Y))) \geq \sum_{i=1}^{k} (ta_u(i) + tr_u(i) |S_u(i)|)$$

GreedyQR has the monotonicity property that $v_i > v_j$ for sources $S_i$ ranked above $S_j$ in permutation $\Pi(Y)$. OnlinePerm performs as well as GreedyQR at worst with no swap happening. Thus, according to the monotonicity property, we have:

$$\frac{T(Q_k(\Pi(Y)))}{T(Q_k(\Pi_{opt}(Y)))} \leq \frac{k \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)}{|Y| \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)}$$

**Corollary 3.** Given $ta_i \ll tr_i |S_i|$ and $tr_i \approx tr_j$ for $\forall S_i, S_j \in Y$, the time cost $T(Q_k(\Pi(Y)))$ of $\Pi(Y)$ by OnlinePerm is 
$$\frac{k \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)}{|Y| \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)}$$
approximate to $T(Q_k(\Pi_{opt}(Y)))$ for TMP.

**Proof:** With the given $ta_i \ll tr_i |S_i|$ and $tr_i \approx tr_j$, the rate of querying any number of tuples with the permutation $\Pi(Y)$ is approximately close to a constant:

$$\frac{k \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)}{|Y| \sum_{i=1}^{u(i)} (ta_u(i) + tr_u(i) |S_u(i)|)} \approx \frac{1}{tr_u(i)}$$

**Algorithm 3:** Online Permutations (OnlinePerm)

**Input:** A query $Q_k$; A permutation $\Pi(Y)$; Statistics

**Input SI:** Intersection Threshold $\theta_{sp}$

**Output:** The new permutation $\Pi(Y)$ for query $Q_k$

1. Set the permutation $\Pi(Y)$ as an empty queue;
2. Set the set $Y_e = Y$;
3. if $\Pi(Y) = \phi$ then
4. \[ [\Pi(Y), Y_e, n_{avg}, v_{avg}] = \]
5. GreedyQR($Q, k, \Pi(Y), Y_e, SI$);
6. end
7. for $i = 1, 2, \ldots, |\Pi(Y)|$ do
8. Fetch the $i$th source $S_i$ from $\Pi(Y)$;
9. \[ [P'(Y), Y', v_{avg}] = \]
10. RePerm($S_i, Q, k, \Pi(Y), Y_e, v_{avg}, SI, \theta_{sp}$);
11. if $v_{avg} > v_{avg}$ then
12. Set $v_{avg} = v_{avg}$ and $\Pi(Y) = \Pi'(Y)$;
13. End
14. end
15. return $\Pi(Y)$;

OnlinePerm sequentially checks the possible swaps between $\forall S_i \in SI$ and $S_j \in Y_e$ having $|S_i \wedge S_j| \geq \theta_{sp}$ from the head to the tail of $\Pi(Y)$. Both the line 7-14 of Algorithm 3 and the line 3-12 of Algorithm 2 runs in $O(l)$ time. The time complexity of GreedyQR (Algorithm 1) is $O(l^2)$. Therefore, OnlinePerm has time complexity $O(l^2)$. Given some restrictions on $\theta_{sp}$, e.g., set $\theta_{sp}(S) = \epsilon$ equal to the maximal value of $|S_r \wedge S_r| / |S_r|$, $\forall S_i \in Y_e$, the time complexity of OnlinePerm can be reduced to $O(l^2)$.

**V. Statistics Collection**

The statistics inputs SI described in the last section should be collected from sources to the querier so as to be taken as inputs for OnlinePerm. The SC component (figure 4) can obtain the access time cost $ta_i$ and the per-tuple transfer time cost $tr_i$ of SI for each source $S_i \in Y$ by generating queries on $S_i$. Then, $ta_i$ and $tr_i$ can be calculated by the time cost of retrieving result tuples, e.g., it takes 850 ms to retrieving 1000 tuples from $S_i$, written as $ta_i + 1000tr_i = 850$ ms. However, SC cannot precisely predict and pre-detect $|\cap Y|$ of SI for a query $Q_k$ given in real time. Alternatively, SC solves this problem in two stages. In Initial-detection stage, SC generates a query $Q_{\hat{Y}}$ that retrieves all tuples from all sources (or samples from all sources), and estimate $|\cap Y|$ for $Q_{\hat{Y}}$. The query result of any other query can be regarded as a subset of the result of $Q_{\hat{Y}}$. Thus, in the Online-detection stage, the estimation of $|\cap Y|$ for a real-time given query $Q_k$ can be derived both from $|\cap Y|$ and online detection results.

**A. Initial-detection**

In the Initial-detection stage, SC generates a query $Q_{\hat{Y}}$ to retrieve all tuples from all sources in $Y$. For the sake of simplicity, we abuse notation and let $|\hat{S}|$ and $|\hat{S}_i|$ denote the number of tuples in and not in $S_i$ for query $Q_{\hat{Y}}$ respectively. Combine all $|\hat{S}|$ or $|\hat{S}_i|$, $i = 1, 2, \ldots, l$, as variables in set $\hat{S}$, e.g., $|\hat{S}_1, \hat{S}_2, \hat{S}_3|$ (for $l = 3$) as a variable denotes the number of tuples in $S_1$ and not in $S_2, S_3$. Also, Let $\hat{S}_i$ represent the set of variables that counts tuples in $q$ sources and not in the other $l - q$ sources. For a variable $\hat{w}_{q+1} \in \hat{\Omega}_{q+1}$, a variable $\hat{w}_q$ is a parent of $\hat{w}_{q+1}$ if $\hat{w}_q$ and $\hat{w}_{q+1}$ are different "in" or "not in" in only one source, e.g., $|\hat{S}_1\hat{S}_2\hat{S}_3|$ in $\hat{\Omega}_1$ is a parent of $|\hat{S}_1\hat{S}_2\hat{S}_3|$ in $\hat{\Omega}_2$. Let $|\hat{S}_i|$ be the ancestor of all variables that consider tuples in $S_i$. As refer to $|\cap \hat{Y}|$, given a permutation $\Pi(\hat{Y})$ of sources, the number of intersection tuples $\cap \hat{S}_i$ for a source $S_i \in Y$ can be estimated by adding up all the $|\hat{S}_i|$ and $|\hat{S}_j|$ variables that consider tuples both in $S_i$ and $S_j$ for any $S_j$ prior to $S_i$ in $\Pi(\hat{Y})$. 

To derive the estimation of $|\cap \hat{Y}|$, SC needs to detect the value of all these variables in $\hat{\Omega}$. The set $\hat{\Omega}$ has a total of $2^l$ variables; the detection complexity of all variables in $\hat{\Omega}$ grows exponentially as $l$ increases. It is impossible to detect all variables when hundreds or even thousands sources exist. Therefore, SC applies the pruning techniques that (1) iteratively add variables to the detection set, (2) iteratively remove variables from the detection set, and (3) estimate the value of variables by Maximum Entropy $\mathcal{L}$.

In detail, SC firstly detects the value of $|\hat{S}_1|$, $|\hat{S}_2|$, ..., $|\hat{S}_l|$ from sources, and start the iterations of $l$ times. Let $\hat{W}_i$ denote the detection set containing variables added in the $i$th iteration. In the $(i+1)$th iteration, SC firstly removes the variables whose value are below a given threshold $\theta_{sc}$ from $\hat{W}_i$, and queries on the sources for the value of the remaining variables in $\hat{W}_i$ after the removal. Then, SC considers the variables in $\hat{\Omega}_{i+1}$. If a variable $\hat{w} \in \hat{\Omega}_{i+1}$ has $\sum \text{Parent}(\hat{w}) > \theta_{sc}$, the value of $\hat{w}$ may also greater than $\theta_{sc}$. SC add such variables like $\hat{w}$ to the detection set $\hat{W}_{i+1}$. Suppose that all variables in $\hat{W}_{i+1}$ have equal weight, and thus according to the principle of Maximum Entropy, SC can estimate the value of these variables in $\hat{W}_{i+1}$ by solving the following MaxEnt problem:

$$\begin{align*}
\text{max:} & \quad -\sum_{\hat{w} \in \hat{W}_{i+1}} \hat{w} \log \hat{w} \\
\text{s.t.} & \quad |\hat{S}_i| = \sum \hat{w}, \hat{w} \in \text{Ancestor}(\hat{S}_i) \& \& \hat{w} \in \hat{W} \\
& \quad \hat{W} = \cup\{\hat{W}_j, j = 1, 2, ..., i+1\}
\end{align*}$$

By solving (10) with Lagrange multipliers $\mathcal{L}$, SC can get the expected value of all variables in $\hat{W}_{i+1}$. When the loop is finished, SC can further estimate $|\cap \hat{Y}|$ for $Q_{\hat{Y}}$ with the value of all variables in $\hat{W}$, $\hat{W} = \cup\{\hat{W}_j, j = 1, 2, ..., l\}$.

**Example 2.** Consider a simple scene of three sources $S_1$, $S_2$ and $S_3$, and assume that all variables to be determined have value greater than $\theta_{sc}$. SC firstly queries on these three sources for the value of $|\hat{S}_1|$, $|\hat{S}_2|$ and $|\hat{S}_3|$. In the 1st iteration, SC solves the MaxEnt problem for $\hat{W}_1$:

$$\begin{align*}
\text{max:} & \quad -\sum_{j=1,2,3} \hat{w}_j \log \hat{w}_j \\
\text{s.t.} & \quad |\hat{S}_1| = \hat{w}_1, |\hat{S}_2| = \hat{w}_2, |\hat{S}_3| = \hat{w}_3
\end{align*}$$

Where we have $\hat{w}_1 = |\hat{S}_1\hat{S}_2\hat{S}_3^c|$, $\hat{w}_2 = |\hat{S}_1\hat{S}_2\hat{S}_3|$, and $\hat{w}_3 = |\hat{S}_1\hat{S}_2\hat{S}_3^c|$. By solving (11), SC can get the estimation of $\hat{w}_1$, $\hat{w}_2$ and $\hat{w}_3$. With the assumption that all variables have value greater than $\theta_{sc}$, no variable is removed from $\hat{W}_1$. In the 2nd iteration, SC detects the value of variable $\hat{w}_1$, $\hat{w}_2$ and $\hat{w}_3$, and solves the MaxEnt problem for $\hat{W}_2$:

$$\begin{align*}
\text{max:} & \quad -\sum_{j=4,5,6,7,8,9} \hat{w}_j \log \hat{w}_j \\
\text{s.t.} & \quad |\hat{S}_1| = \hat{w}_1 + \hat{w}_4 + \hat{w}_5, |\hat{S}_2| = \hat{w}_2 + \hat{w}_6 + \hat{w}_7, |\hat{S}_3| = \hat{w}_3 + \hat{w}_8 + \hat{w}_9
\end{align*}$$

Where we have $\hat{w}_4 = |\hat{S}_1\hat{S}_2\hat{S}_3^c|$, $\hat{w}_5 = |\hat{S}_1\hat{S}_2\hat{S}_3^c|$, $\hat{w}_6 = |\hat{S}_1\hat{S}_2\hat{S}_3^c|$, $\hat{w}_7 = |\hat{S}_1\hat{S}_2\hat{S}_3|$, $\hat{w}_8 = |\hat{S}_1\hat{S}_2\hat{S}_3^c|$, and $\hat{w}_9 = |\hat{S}_1\hat{S}_2\hat{S}_3^c|$. In the 3rd iteration, SC solves the MaxEnt problem for $\hat{W}_3 = \{\hat{S}_1\hat{S}_2\hat{S}_3\}$. Given a permutation $S_1S_2S_3$, the number of intersection tuples for any source could be estimated, e.g. $|\cap \hat{S}_1| = |\hat{S}_1\hat{S}_2\hat{S}_3| + |\hat{S}_1\hat{S}_2\hat{S}_3| + |\hat{S}_1\hat{S}_2\hat{S}_3|$.

The formal description of Initial-detection algorithm is omitted here due to space limit. Initial-detection algorithm runs loop of $l$ times, each time with $|\hat{W}_i|$ variables and $l$ constraints for the MaxEnt problem, $i = 1, 2, ..., l$.

**B. Online-detection**

After the Initial-detection stage, an estimation of $|\cap \hat{Y}|$ for $Q_{\hat{Y}}$ has already been established. Then, a permutation $\Pi(\hat{Y}) = S_{(1)}S_{(2)}...S_{(l)}$ of all sources can be constructed by OnlinePerm based on $|\cap \hat{Y}|$. Upon receiving a real-time query $Q$ of $k$ tuples, SC start the Online-detection stage, and derives the estimation of $|\cap Y|$ from $|\cap \hat{Y}|$ on-the-fly simultaneously with the query execution.

In the first sub-stage, the number of tuples $|S_{(i)}|$, $i = 1, 2, ..., l$ for $Q_{k}$ are detected sequentially following the permutation $\Pi(\hat{Y})$. When receiving partial results $|S_{(1)}|$, $|S_{(2)}|$, ..., $|S_{(q)}|$, SC estimates the number of tuples for other sources:

$$|S_{(i)}| = \frac{\hat{S}_{(i)}}{q} \sum_{j=1}^{q} \frac{S_{(j)}}{q}, i = q + 1, q + 2, ..., l$$

Rewrite the MaxEnt problem as follow:

$$\begin{align*}
\text{max:} & \quad -\sum_{w \in W} w \log w \\
\text{s.t.} & \quad |S_{(i)}| = \sum w, w \in \text{Ancestor}(\hat{S}_{(i)}) \& \& w \in W \\
& \quad W = \cup\{\hat{W}_j, j = 1, 2, ..., l\}
\end{align*}$$

The MaxEnt problem in the Online-detection stage considers all variables introduced in the Initial-detection stage. By solving (14), SC can get the expected value of all variables in $W$, and thus estimate $|\cap Y|$ for any given permutation $\Pi(Y)$.

In the second sub-stage, all the results of $|S_{(1)}|$, $|S_{(2)}|$, ..., $|S_{(i)}|$ have been received. SC sorts all variables $w \in W$ by their expected value $|w - \hat{w}|$ in descending order, where $\hat{w}$ denotes the value of $w$ estimated in the Initial-detection stage. Then, SC sequentially detects the value of these variables in $W$ following this order. Upon receiving partial results of detection, SC resolves the MaxEnt problem of (14) for the estimation of $|\cap Y|$. The Online-detection is terminated when the query execution of $Q_k$ is finished or all the results of detection have been received.

The formal description of Online-detection algorithm is omitted here due to space limit. Online-detection algorithm resolves the MaxEnt problem with at most $|W|$ times; in the $i$th time, estimate the expected value of no more than $|W| - i + 1$ variables.
VI. ONLINE QUERY STRATEGY

The Online Query Framework is shown in figure 6. It applies the dynamic strategy that enables the execution of query on sources and the improvement of source permutation to work in parallel.

At the beginning, SC generates a query $\hat{Q}_Y$, that retrieves all tuples from all sources (or samples from all sources), and start the Initial-detection process to probe and estimate $|\cap Y|$. Then, SP applies OnlinePerm algorithm to construct a permutation $P(\hat{Y})$ for all sources based on the estimation of $|\cap Y|$.

Upon receiving a real-time query $Q_k$, the processes in SC, SP and QE start to work simultaneously. In detail, SC starts the Online-detection process to derive the estimation of $|\cap Y|$ from both the results of online detection and the estimation of $|\cap \hat{Y}|$. SP repeatedly runs the OnlinePerm algorithm on un-queried sources with the continuously new estimation of $|\cap Y|$ provided by SC; at the end of each run, the OnlinePerm algorithm writes the newly constructed permutation $P(Y)$ to the shared memory for the query $Q_k$. Simultaneously, QE reads the permutation $P(Y)$ from the shared memory, and starts query threads to retrieve the result tuples sequentially from sources following $P(Y)$. When the querier has already received $k$ tuples or all sources have been queried, QE sends a signal to terminate the running process in SC and SP.

VII. EXPERIMENT RESULTS

We conduct experiments on independent sources that are self-controlled and only provide query interface for others. Each source consists of tuples of educational institutions, as shown in Figure 7 crawled from web sites with a random start seed. In experiments, we divide tuples into two sets: $E_1$ and $E_2$, having $E = E_1 \cup E_2$ and $E_1 \cap E_2 = \emptyset$. Each tuple is either belong to $E_1$ or $E_2$. We use the query $\hat{Q}$ of $\textsc{Select} *$ FROM $E$ for the Initial-detection stage of SC and the query $Q$ of $\textsc{Select} top k$ tuples FROM $E_1$ to evaluate the time cost of implemented algorithms.

For evaluation, we implemented eight algorithms:

- Random: Randomly choose a permutation of sources.
- MaxT: Select the source $S$ with the maximal tuples $|S|$ each time without considering intersection between sources.
- MaxRT: Select the source $S$ with the maximal residual tuples $|S| - |\cap S|$ each time that considers the intersection compared with MaxT.
- MinT: Select the source $S$ with the minimal per-tuple retrieve time $\frac{ta + tr|S|}{|S|}$ each time without considering the intersection.
- MinRT: Select the source $S$ with the minimal residual per-tuple retrieve time $\frac{ta + tr|S|}{|S| - |\cap S|}$ each time that considers the intersection compared with MinT.
- SeqPerm: apply the sequential strategy that starts the query execution until the finish of RePerm (Algorithm 3). SeqPerm is with additional time cost of permutation construction, but may have a better permutation than OnlinePerm.
- OnlinePerm: apply the online strategy that let all the components work in parallel. OnlinePerm does not consider the sources have been queried, and only construct permutation for un-queried sources dynamically without waiting for the finish of RePerm.
- FullKnowledge: apply the permutation constructed by OnlinePerm with precise intersection statistics $|\cap Y|$ as input. Its performance can be considered as the upper bound can be achieved although the permutation may not be optimal (Theorem 3).

All the algorithms were implemented in Java JDK 1.7, and ran experiments on a Data Integration System (DIS) of our implementation. DIS can manipulate remote relational databases or shared folders to create a new source. DIS is built on 4 physical machines. By default, DIS has 2035 sources with a total of 501760 tuples in these machines, and each tuple is limited with the maximal size of 120B. Among these tuples, there are 24860 distinct tuples in all. We observed that the access time cost of any source in DIS is in $[477, 2350]$ms, and the per-tuple transfer time cost is in $[0.02, 0.42]$ms. The querier was implemented on a Windows 7 machine with 2.3GHz Intel Core i7 CPU, 8GB RAM and Gigabit Ethernet Controller. Both the querier and DIS are in the same LAN. For simplicity, we abuse the notation and let $|Y_1|, |Y_2|$ and $|Y|$ denote the number of tuples in $E_1, E_2$ and $E$ respectively. By default, we set $|Y_1| = |Y_2| = 0.5|Y| = 12430, k = 0.8|Y_1|$, the intersection threshold $\theta_{sp} = 0.05$, the detection threshold $\theta_{ac} = 0.005$, one thread running for the query execution and one thread running for the Online-detection process of $Q$ in the querier. We next evaluate the query time cost of implemented algorithms under conditions of different factors.

A. Varying $k$

We firstly compared the algorithms under the condition of varying $k$ by default settings. The row 2-5 of Table 4
show the results of query time cost of the algorithms for $k = 0.2|Y_1|$, 0.4$|Y_1|$, 0.6$|Y_1|$ and 0.8$|Y_1|$ respectively. We have the following observations.

First, Random performs worst among these algorithms. By considering the number of tuples in sources, MaxT has slightly less time cost than Random; By further considering the number of intersection tuples between sources, MaxRT performs better than MaxT. By selecting the sources with per-tuple time cost, MinT is faster than MaxT, and by considering the intersection in per-tuple time cost, MinRT has even faster query rate than MinT.

Second, SeqPerm has significantly initial time cost for permutation construction and performs worse than MinRT at the beginning. As shown in Table I, SeqPerm has approximately 5.5s (or seconds) and 3.9s more time cost than MinRT for $k = 0.2|Y_1|$ and $k = 0.4|Y_1|$ respectively. Then, SeqPerm quickly catches up MinRT and performs better than MinRT; SeqPerm has approximately 0.9s and 0.5s less time cost than MinRT for $k = 0.6|Y_1|$ and $k = 0.8|Y_1|$ respectively. The results of SeqPerm show the effectiveness of RePerm algorithm that constructs a better permutation than MinRT.

Third, OnlinePerm has roughly the same performance with MinRT at the beginning, and as $k$ increases, OnlinePerm has apparently less time cost than MinRT. This is because that the effect of intersection between sources is little or non-existent when the number of tuples, $k$, to be retrieved is small, and the effect of intersection appears when $k$ is increased. As shown in Table I, OnlinePerm and MinRT have approximately time cost of 10.7s and 11.1s for $k = 0.2|Y_1|$ respectively, and OnlinePerm takes approximately 85.1s to retrieve $k = 0.8|Y_1|$ tuples while MinRT takes approximately 96.7s.

Finally, as can be seen from Table I, OnlinePerm has stable less time cost (4-6s) than SeqPerm for $k = 0.2|Y_1|$, 0.4$|Y_1|$, 0.6$|Y_1|$ and 0.8$|Y_1|$ respectively, which shows that the parallel execution of processes in SP and QE component would not reduce (or has little effect on) the quality of the permutation constructed by RePerm; OnlinePerm has little more time cost (0-s) than FullKnowledge, which shows the efficiency of processes that online collect statistics for source permutation in SC component.

### B. Varying Query Threads

We compared the algorithms on varying query threads. The row 6-9 of Table I show the results of query time costs of the algorithms for $k = 0.8|Y_1|$ when running 2, 3, 4 and 5 query threads at the same time respectively. As can be seen, when more query threads are running, (1) the query time cost of the algorithms decrease fairly fast, (2) the decrease rate of query time cost slows down we usually observed in parallel system, and (3) the advantage of OnlinePerm on the performance becomes less apparent, e.g., OnlinePerm spends approximately 18.2s on query execution than 19.7s by MaxRT when 5 query threads are running. The performance of OnlinePerm reveals that the benefit of Online-detection for statistics collection is banalanced by increased running query threads.

### C. Varying Sources

We compared the algorithms on varying sources, and let the number of sources be 2000, 3000, 4000 and 5000 respectively by disabling or creating sources in DIS for evaluation. Then, we kept the number of distinct tuples and the total number of tuples as default settings by moving tuples between sources. The evaluation results of varying number of sources for $k = 0.8|Y_1|$ are shown in row 10-13 of Table I. As can be seen, with more sources involved, (1) the query time of the

| Condition | Algorithm | Random | MaxT | MaxRT | MinT | MinRT | SeqPerm | OnlinePerm | FullKnowledge |
|-----------|-----------|--------|------|-------|------|-------|---------|------------|---------------|
| $k = 0.2|Y_1|$ | 38751.6 | 30478.7 | 26850.1 | 13807.9 | 11058.3 | 16617.4 | 10735.3 | 10528.6 |
| $k = 0.4|Y_1|$ | 78872.4 | 65944.0 | 60727.7 | 32861.0 | 27899.4 | 31831.5 | 25709.8 | 24147.1 |
| $k = 0.6|Y_1|$ | 133642.2 | 120663.7 | 109618.4 | 59887.5 | 52291.0 | 51415.3 | 47623.3 | 43979.9 |
| $k = 0.8|Y_1|$ | 236555.6 | 209363.2 | 195653.2 | 114498.3 | 96680.0 | 90179.0 | 85064.9 | 81170.8 |

1. All values are in milliseconds, or ms.
algorithms increase, (2) the query time cost of OnlinePerm is less than other algorithms, and (3) the query time cost of OnlinePerm increases linearly and is approximate to the time cost of FullKnowledge, which shows the scalability of our online query system.

D. The Effect of Pruning Techniques

We applied the pruning techniques introduced in Section V for estimation of $|\cap Y_1|$ based on the statistics collected by the SC component. With huge number of variables removed by the pruning techniques, SC only need to solve the MaxEnt problem of 5137 variables at the beginning of the Online-detection stage, and the number of variable becomes less with the execution of Online-detection process. By converting the MaxEnt problem to a sparse system of linear equations, SC estimated the value of $Y$ with the execution of Online-detection process. By converting the MaxEnt problem to a sparse system of linear equations, SC estimated the value of $Y$ with the execution of Online-detection process. By converting the MaxEnt problem to a sparse system of linear equations, SC estimated the value of $Y$ with the execution of Online-detection process. By converting the MaxEnt problem to a sparse system of linear equations, SC estimated the value of $Y$ with the execution of Online-detection process.

The Error of Initial-detection Statistics

We compared the algorithms on errors of Initial-detection statistics. The statistics estimation for $\mathcal{Q}$ is more accurate with a higher $|Y_1|$ and we set $|Y_1| = 0.2|Y|, 0.4|Y|, 0.6|Y|$ and $0.8|Y|$ respectively for evaluation. The evaluation results for $k = 0.8|Y_1|$ are shown in row 14-17 of Table I. As can be observed, with higher $|Y_1|$, (1) the query time cost of the algorithms except random decrease, and especially (2) the decrease rate of query time cost of OnlinePerm is small, e.g., OnlinePerm spends approximately 84.4s for $|Y_1| = 0.6|Y|$ and 83.1s for $|Y_1| = 0.8|Y|$. The result of OnlinePerm shows that OnlinePerm has stable performance on errors of Initial-detection statistics.

F. The Overhead of Online-detection

We compared the algorithms on overheads of Online-detection by adding cycle time and setting the detection time cost to be 1.2, 1.4, 1.6 and 1.8 times of original detection time cost. The results are shown in row 18-21 of Table I. As can be observed from the results, with higher overhead of Online-detection, (1) the query time cost of the algorithms except random increase, and (2) OnlinePerm has a high increase rate of query time cost although it still performs better than other algorithms. The results reveal that it is getting harder to gain benefit from Online-detection when the detection overhead is increasing. In this case, more running threads for Online-detection are suggested to balance the effect of increased overhead.

G. Summary

We evaluated our online scheduling strategy under the condition of various factors.

- Varying $k$: OnlinePerm is the fastest algorithm among all these algorithms; this evaluation result shows the efficiency of our strategy that enables SP, SC and QE working in parallel.
- Varying query threads: OnlinePerm still performs best among all these algorithms, but the benefit of Online-detection of SC is balanced by increased running threads.
- Varying sources: The query time cost of OnlinePerm increases linearly and least among all the algorithms when more sources are involved; this evaluation result shows that our strategy is scalable.
- The effect of pruning techniques: By applying the pruning techniques, SC can efficiently estimate the statistics of sources.
- The error of initial-detection statistics: OnlinePerm suffers a low performance degradation when the error of Initial-detection increases; this evaluation result shows that our strategy is efficient and robust.
- The overhead of online-detection: Although OnlinePerm still perform best among all these algorithms, it has a high performance degradation when detection overhead increases. This result suggests more detection threads of SC to balance the effect of increased detection overhead.

VIII. Conclusion and Future Work

We address the Time-Cost Minimization Problem (TMP) and propose the online scheduling strategy in this paper. The architecture of online query scheduling mainly contains three components of Source Permutation (SP), Statistics Collection (SC) and Query Execution (QE). We prove that it is NP-complete to construct a optimal permutation of sources and propose OnlinePerm algorithm that considers the effect of query rate and residual tuples for SP. We present a two-stage detection mechanism and apply pruning techniques to avoid the exponential number of variables estimation for SC. By applying the online scheduling strategy, SP, SC and QE work in parallel to reduce the total time cost for the query. The experiment results show the efficiency and scalability of our scheduling strategy.

In this paper, we simplify the redundant problem by only considering the repetitive data tuples between sources. In the future, we would consider the time cost of data fusion between partially overlapping tuples during online query scheduling.

References

[1] Aggregator applications. http://en.wikipedia.org/wiki/Aggregator
[2] J. Bleiholder, S. Khuller, F. Naumann, L. Raichudd, and Y. Wu. Query planning in the presence of overlapping sources. In EDBT, 2006.
[3] N. N. Dalvi, A. Machanavajjhala, and B. Pang. An Analysis of Structured Data on the Web. In PVLDB, 5(7):680-691, 2012.
[4] X. L. Dong, L. Berti-Equille, and D. Srivastava. Data Fusion: Resolving Conflicts from Multiple Sources. Handbook of Data Quality, pp.293-318. Springer-Verlag, 2013.
[5] D. Florescu, D. Koller, and A. Y. Levy. Using probabilistic information in data integration. In VLDB, 1997.
[6] E. T. Jaynes. Information Theory and Statistical Mechanics. In Physical Review Series II, 106(4):620-630, 1957.
[7] Z. Nie, S. Kambhatlapati, and U. Nambiar. Effectively mining and using coverage and overlap statistics for data integration. In TKDE, 17(5):638-651, 2005.
[8] M. Sallom, X. L. Dong, D. Srivastava, and V. J. Tsotras. Online Ordering of Overlapping Data Sources. In PVLDB, 7(3):133-144, 2013.
[9] A. D. Sarma, X. L. Dong, and A. Halevy. Data integration with dependent sources. In EDBT, 2011.
[10] J. Skillings, and R. K. Bryan. Maximum entropy image reconstruction: general algorithm. In Monthly Notices of the Royal Astronomical Society 211, pp.111-124, 1984.
[11] B. Trushkowsky, T. Kraska, M. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In ICDE, 2013.
[12] V. Vassalos, and Y. Papakonstantinou. Using knowledge of redundancy for query optimization in mediators. In AAAI Workshop on AI and Info. Integration, 1998.