Parallel Data Access for Multiway Rank Joins

Adnan Abid and Marco Tagliasacchi

Dipartimento di Elettronica e Informazione – Politecnico di Milano,
Piazza Leonardo da Vinci, 32 – 20133 Milano, Italy
{abid,tagliasa}@elet.polimi.it

Abstract. Rank join operators perform a relational join among two or more relations, assign numeric scores to the join results based on the given scoring function and return K join results with the highest scores. The top-K join results are obtained by accessing a subset of data from the input relations. This paper addresses the problem of getting top-K join results from two or more *search* services which can be accessed in parallel, and are characterized by non negligible response times. The objectives are: *i*) minimize the time to get top-K join results. *ii*) avoid the access to the data that does not contribute to the top-K join results.

This paper proposes a multi-way rank join operator that achieves the above mentioned objectives by using a score guided data pulling strategy. This strategy minimizes the time to get top-K join results by extracting data in parallel from all Web services, while it also avoids accessing the data that is not useful to compute top-K join results, by pausing and resuming the data access from different Web services adaptively, based on the observed score values of the retrieved tuples. An extensive experimental study evaluates the performance of the proposed approach and shows that it minimizes the time to get top-K join results, while incurring few extra data accesses, as compared to the state of the art rank join operators.

Keywords: rank joins, rank queries, score guided data pulling, top-K queries.

1 Introduction

Rank join operators have a widespread applicability in many application domains. Hence, a set of specialized rank join operators have been recently proposed in the literature [1][4][6][8][9]. These operators are capable of producing top-K join results by accessing a subset of data from each source, provided the score aggregation function is monotone, and the data retrieved from each source is sorted in descending order of score.

As an illustrative example, consider a person who wants to plan his visit to Paris by searching for a good quality hotel and a restaurant, which are situated close to each other and are highly recommended by their customers. This can be accomplished by extracting information from suitable data sources available on the Web and merging the information to get the top rated resultant combinations, as contemplated in Search Computing [3]. The Web services, e.g. `Yahoo!`

S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.): ICWE 2011, LNCS 6757, pp. 44–58, 2011.

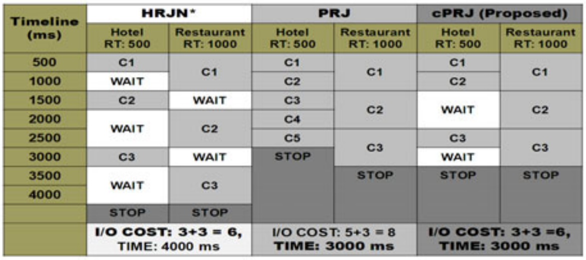
Local or yelp.com, can be used to find the places of interest in a city. The data can be processed to produce the top-K scoring join results of hotels and restaurants. A sample rank query based on the above example is the following:

SELECT h.name, r.name, 0.6*h.rating+0.4*r.rating as score
FROM Hotels h, Restaurants r
WHERE h.zip = r.zip AND h.city= 'Paris' AND r.city = 'Paris'
RANK BY 0.6*h.rating+0.4*r.rating

Motivation: The recent solutions to rank join problem [5][7][12] focus on providing instance optimal algorithms regarding the I/O cost. The I/O cost is a quantity proportional to the overall number of fetched tuples. So these algorithms minimize the total number of tuples to be accessed in order to find the top-K join results. *Hash Rank Join* (HRJN*) [7] is an instance optimal algorithm in terms of I/O cost and it introduces a physical rank join operator. This algorithm has been further improved in [5] and [12]. Indeed, this optimization of the I/O cost helps reducing the total time to compute the top-K join results as well, yet total time can be further reduced for the following reason: these I/O optimal algorithms access data from the data sources in a serial manner, i.e. they access data from one source, process it and then fetch the data from the next *most suitable* source. The latter is selected based on a *pulling strategy*, which determines the source to be accessed to, in order to minimize the I/O cost. However, in the context of using Web services as data sources, data processing time is found to be negligible as compared to data fetching time. So, most of the time is spent in waiting for retrieving the data. Therefore, an alternative approach that extracts data from all data sources in parallel should be used in order to reduce the data extraction time from all sources by overlapping the waiting times. This calls for a parallel data access strategy.

A simple parallel strategy keeps on extracting data from each Web service in parallel until top-K join results can be reported. We call this strategy *Parallel Rank Join* (PRJ). As an illustrative example, assume that we can extract top 10 join results from 2 different Web services after fetching 3 data pages from each Web service. Figure 1 shows the behaviour of both HRJN* and PRJ. It can be observed that both HRJN* and PRJ approaches have shortcomings: HRJN* takes a large amount of time to complete, whereas, PRJ costs more in terms of I/O as it may retrieve unnecessary data (e.g. C4 and C5). This requires the design of a rank join operator that is specifically conceived to meet the objectives of getting top-K join results quickly and restricting access to unwanted data, when using Web services or similar data sources.

As a contribution we propose a *Controlled Parallel Rank Join* (cPRJ) algorithm that computes the top-K join results from multiple Web services with a *controlled* parallel data access which minimizes both total time, and the I/O cost, to report top-K join results. In Section 2 we provide the preliminaries. The algorithm is explained in Section 3. A variant of the algorithm is presented in Section 4. The experiments and results are discussed in Section 5, and related work and conclusion are presented in Sections 6 and 7, respectively.



Timeline (ms)	HRJN*		PRJ		cPRJ (Proposed)	
	Hotel RT: 500	Restaurant RT: 1000	Hotel RT: 500	Restaurant RT: 1000	Hotel RT: 500	Restaurant RT: 1000
500	C1	C1	C1	C1	C1	C1
1000	WAIT		C2		C2	
1500	C2	WAIT	C3	C2	WAIT	C2
2000	WAIT	C2	C4			
2500			C5	C3	C3	C3
3000	C3	WAIT	STOP		WAIT	
3500	WAIT	C3		STOP	STOP	STOP
4000						
	STOP	STOP				
	I/O COST: 3+3 = 6, TIME: 4000 ms		I/O COST: 5+3 = 8 TIME: 3000 ms		I/O COST: 3+3 =6, TIME: 3000 ms	

Fig. 1. Serial Data Access of HRJN* vs Parallel Data Access

2 Preliminaries

Consider a query Q whose answer requires accessing a set of Web services $S_1, ..., S_m$, that can be wrapped to map their data in the form of tuples as in relational databases. Each tuple $t_i \in S_i$ is composed of an identifier, a join attribute, a score attribute and other named attributes. The tuples in every Web service are sorted in descending order of score, where the score reflects the relevance with respect to the query. Let $t_i^{(d)}$ denote a tuple at position d of S_i. Then $\sigma(t_i^{(d)}) \geq \sigma(t_i^{(d+1)})$, where $\sigma(t_i)$ is the score of the tuple t_i. Without loss of generality, we assume that the scores are normalized in the [0,1] interval.

Each invocation to a Web service S_i retrieves a fixed number of tuples, referred to as chunk. Let (CS_i) denote the *chunk size*, i.e. the number of tuples in a chunk. The chunks belonging to a Web service are accessed in sequential order, i.e. the $c - th$ chunk of a Web service will be accessed before $(c + 1) - th$ chunk. Each chunk, in turn, contains tuples of S_i sorted in descending order of score. Furthermore, S_i provides one chunk of tuples in a specified time, which is referred to as its *average response time* (RT_i). Let $t = t_1 \bowtie t_2 \bowtie ... \; t_m$ denote a join result formed by combining the tuples retrieved from the Web services, where t_i is a tuple that belongs to the Web service S_i. This join result is assigned an aggregated score based on a monotone score aggregation function, $\sigma(t) = f(\sigma(t_1), \sigma(t_2), .., \sigma(t_m))$. The join results obtained by joining the data from these Web services are stored in a buffer S_{result} in descending order of their aggregate score.

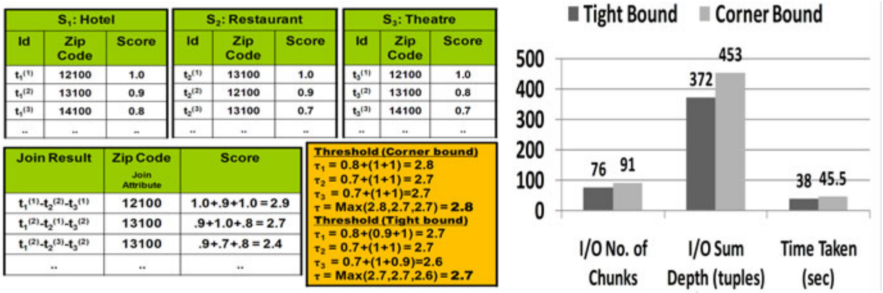
2.1 Bounding Schemes

Let τ_i denotes the local threshold of a Web service S_i which represents an upper bound on the possible score of a join result that can be computed by joining any of the unseen tuples of S_i to either seen or unseen data of the rest of the Web services. The global threshold τ of all the Web services is the maximum among the local thresholds i.e. $\tau = \max\{\tau_1, \tau_2, ..., \tau_m\}$.

The local threshold is updated with each data access to the corresponding Web service. Whereas, the global threshold is updated after every data access, independent of the accessed Web service. The bounding scheme is responsible for computing τ, which represents an upper bound on the scores of possible join results, which can be formed by the unseen data. Thus, it helps in reporting the identified join results to the user. Let K denote the number of join results for which $\sigma(t) \geq \tau$, then these can be guaranteed to be the top-K. Figure 2(a) illustrates an example in which the global threshold is computed based on two possible bounding schemes based on the snapshot of execution, when all sources have fetched three tuples. The join predicate is the equality between the zip code attribute. These two bounding schemes *corner bound* and *tight bound* are further discussed below.

Corner Bound: The local threshold for a Web service S_i is calculated by considering the score of last seen tuple of S_i and maximum possible scores for the rest of the Web services. As an example, in Figure 2(a), the local threshold for S_1 is $\tau_1 = f(\sigma(t_1^{(3)}), \sigma(t_2^{(1)}), \sigma(t_3^{(1)})) = f(0.8, 1.0, 1.0) = 2.8$, assuming a simple linear score aggregation function. There is a drawback in using the threshold as computed by means of the *corner bound.* Indeed, it implicitly assumes that the first tuples of all the Web services formulate a valid join result, which may or may not be the case. Therefore, when more than two Web services are involved in a join, if the first tuples of all the Web services do not satisfy the join predicate, then the computed value of the *corner bound* threshold is not tight, in the sense that it might not be possible to from join results with unseen data that achieves that score. Note that HRJN* adopts a *corner bound* [7].

Tight Bound: It is possible to compute τ as a *tight bound* on the aggregate score of unseen join results [12]. The local threshold for a Web service S_i can be calculated by considering the score of the last seen tuple from S_i and the score of the partial join result, PJ_i, with maximum *possible* score which is formed by the rest of the Web services. Let $\mathcal{N}_i = \{i_1, ..., i_n\}$ denote a subset of $\{1, ..., m\}$ which does not contain the index i of Web service S_i, and $n = |N|$, $0 \leq n < m$. There can be 2^{m-1} such distinct subsets. We find the join result with maximum possible score for each distinct subset $\mathcal{N}_i^j$, where $0 < j \leq 2^{m-1}$, and store it in $PJ(\mathcal{N}_i^j)$. The join results for a particular subset $\mathcal{N}_i^j$ are computed by joining the seen tuples from the Web services whose indices are in $\mathcal{N}_i^j$ and completing the join result with a tuple from the rest of the Web services i.e. $N - \mathcal{N}_i^j$, whose score is equal to the score of last seen tuple in the respective Web service. In this way, the join result in $PJ(\mathcal{N}_i^j)$ has one tuple from every Web service. The local threshold τ_i for S_i is computed as max $\sigma(PJ(\mathcal{N}_i^j)), 0 < j \leq 2^{m-1}$. The maximum of all local thresholds is considered as global threshold. Further optimizations in the computation of the *tight* bound are discussed in [5]. When there are only two Web services [11], or the top scoring tuples in each service contribute to $PJ(\mathcal{N}_i^j)$, the *tight* bound and *corner* bound are equivalent. Figure 2(b) shows the average gain in terms of I/O cost and fetch time while using *tight* bound over *corner* bound using HRJN*, averaged over 10 different data sets.



S_1: Hotel		
Id	Zip Code	Score
$t_1^{(1)}$	12100	1.0
$t_1^{(2)}$	13100	0.9
$t_1^{(3)}$	14100	0.8
..	..	..

S_2: Restaurant		
Id	Zip Code	Score
$t_2^{(1)}$	13100	1.0
$t_2^{(2)}$	12100	0.9
$t_2^{(3)}$	13100	0.7
..	..	..

S_3: Theatre		
Id	Zip Code	Score
$t_3^{(1)}$	12100	1.0
$t_3^{(2)}$	13100	0.8
$t_3^{(3)}$	14100	0.7
..	..	..

Join Result	Zip Code Join Attribute	Score
$t_1^{(1)}$-$t_2^{(2)}$-$t_3^{(1)}$	12100	1.0+.9+1.0=2.9
$t_1^{(2)}$-$t_2^{(1)}$-$t_3^{(2)}$	13100	.9+1.0+.8=2.7
$t_1^{(2)}$-$t_2^{(3)}$-$t_3^{(2)}$	13100	.9+.7+.8=2.4
..	..	..

Threshold (Corner bound)
$\tau_1 = 0.8+(1+1) = 2.8$
$\tau_2 = 0.7+(1+1) = 2.7$
$\tau_3 = 0.7+(1+1)=2.7$
$\tau = \text{Max}(2.8,2.7,2.7) = \mathbf{2.8}$
Threshold (Tight bound)
$\tau_1 = 0.8+(0.9+1) = 2.7$
$\tau_2 = 0.7+(1+1) = 2.7$
$\tau_3 = 0.7+(1+0.9)=2.6$
$\tau = \text{Max}(2.7,2.7,2.6) = \mathbf{2.7}$

(a) (b)

Fig. 2. (a) An example scenario, *tight* and *corner* bounding schemes. (b) Gain in I/O and time by using *tight bound* over *corner bound* with 4 Web services and K=20.

2.2 Data Pulling Strategy

The data pulling strategy provides a mechanism to choose the *most suitable* data source to be invoked at a given time during the execution [7]. The pulling strategy can be as simple as a round-robin strategy. HRJN* which focuses only on the optimization of the I/O cost, adopts a pulling strategy whereby the next service to be invoked is the one whose local threshold is equal to τ, the ties are broken by choosing the service which has extracted lesser number of tuples. The intuition of this pulling strategy is to keep all local thresholds as close as possible, which, due to monotonicity, is only possible by extracting the data from the data source with the highest local threshold. But the problem with this pulling strategy is that it takes longer time as shown in Figure 1. The objective of our work is to propose a pulling strategy that exploits the possibility of parallel access to the services. Such a strategy aims not only at minimizing the I/O cost, but also minimizing the time to fetch the data and hence, the time to report top-K join results. Our data pulling strategy is explained below in Section 3.1.1.

3 Methodology

3.1 Proposed Data Pulling Strategy

We stress on such a data pulling strategy which extracts data from all Web services in parallel. A naïve parallel pulling strategy, PRJ, keeps on extracting data from every data source till its respective local threshold becomes lesser or equal to the score of the *then* seen K-th join result. Figure 1 shows the comparison of different data pulling strategies. It shows that the I/O optimized HRJN* strategy has least I/O cost, but it takes more time to get top-K join results. Whereas, PRJ is only concerned with reducing the time to get top-K join results and it may result the extraction of unwanted data. This extraction of unwanted data is possible if a Web service stops well before the others, that is, its local threshold has reached below the score of the *then* top-K-th join result in the output buffer S_{result}. In this case, there is a possibility that the other Web services having higher local thresholds produce join results with better aggregate score values, and terminate with an even higher local threshold. Resultantly, the Web service which stops earlier incurs extra data fetches. Therefore, in case of m Web services maximum $m-1$ Web services may terminate earlier than the m-th Web service. Our proposed data pulling strategy extracts data from all the data sources in *controlled* parallel manner, the parallel data access helps minimizing the time to get top-K join results. Whereas, the I/O cost is minimized by pausing and resuming data extraction from the Web services. The pausing and resuming of data extraction from a Web service with lower local threshold, are performed on the basis of estimating the time to bring the local threshold of other Web services with higher local thresholds below or equal to its local threshold. This is explained in the Section 3.1.2. We use *tight* bounding scheme to compute the threshold values.

3.1.1 State Machine

In order to refrain from accessing the data that do not contribute to the top-K join results every Web service is controlled by using a state machine shown in Figure 3. The Web services are assigned a particular state after the completion of the processing of data fetched from any Web service. The *Ready* state means that the data extraction call should be made for this Web service. It is also the starting state for each Web service. A Web service S_i is put into $Wait$ if we can fetch more data from any other Web service S_j and still its local threshold τ_j, will remain greater than or equal to τ_i. The *Stop* state means that further data extraction from this Web service will not contribute to determining the top-K join results. Lastly, the $Finish$ state means that all the data from this Web service has been retrieved. The *Stop* and $Finish$ states are the end states of the state machine. The difference between PRJ and the proposed cPRJ is that PRJ does not have $Wait$ state, whereas, cPRJ controls the access to the unwanted data by putting the Web services into $Wait$ state. On retrieving a chunk of tuples from Web service S_i the following operations are performed in order:

1. Its local threshold τ_i is updated and it is also checked if the global threshold τ also needs to be updated.
2. New join results are computed by joining the recently retrieved tuples from S_i with the tuples already retrieved from all other Web services.
3. All join results are stored in the buffer S_{result} in descending order of score. The size of the buffer S_{result} is bound by the value of K. All join results having aggregated score above τ are reported to the user.
4. The state for S_i is set using *setState* function shown in Figure 4(a). If S_i has extracted all its data then it is put to $Finish$ state and τ_i is set to 0.

Apart from this the following operations are also performed:

1. Every Web service S_i, which is not in *Stop* or $Finish$ state, is checked and is put into *Stop* state, if $\sigma(t^{(K)}_{result}) \geq \tau_i$.

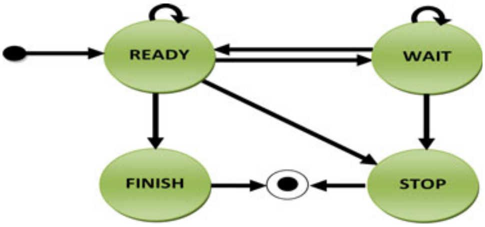


Fig. 3. The state machine according to which each Web service is manipulated

2. A Web service S_i that is in $Wait$ state is put to *Ready* state, if there is no other Web service S_j which is in *Ready* state and τ_j is greater than τ_i, and S_j needs more than one chunk to bring τ_j lower than τ_i, and the minimum time needed to bring τ_j less than τ_i is greater than RT_i.

The state transitions are exemplified below in Section 3.1.3.

3.1.2 Time to Reach (ttr)

Data pulling strategy issues the data extraction calls by analyzing the local thresholds of the Web services. Particularly, the decisions to put a service from *Ready* to $Wait$, and $Wait$ to *Ready* state are based on the computation of time to reach (*ttr*). Therefore, in order to clearly understand these state transitions we need to understand the computation of *ttr*. On completion of a data fetch from Web service S_i we identify all the Web services which are in *Ready* state and have higher local threshold value than τ_i, and put them in a set J. For each Web service S_j, in set J, we compute *time to reach*, (ttr_j), which is the time that S_j will take to bring τ_j below τ_i. The highest value of ttr_j is considered as *ttr* for Web service S_i. If *ttr* is greater than RT_i then S_i is put into $Wait$ state, otherwise, it remains in *Ready* state.

The estimation of *ttr* involves the calculation of decay in score for the Web service S_j. We use Autoregressive Moving Average forecasting method [2] for the calculation of score decay. After estimating the unseen score values we can compute the total number of tuples needed to bring the τ_j lower than the value of τ_i. This number is then divided by the *chunk size* of S_j i.e. CS_j, to get the number of *chunks* to bring the threshold down. If number of *chunks* are less than one, i.e. the after getting the data from the currently extracted chunk τ_j will fall below τ_i, then ttr_j is set to 0. Otherwise, number of *chunks* are multiplied by RT_j, and the elapsed time ET_j, the time since the last data extraction call is issued for S_j is subtracted i.e. $ttr_j = (chunks \times RT_j) - ET_j$.

3.1.3 State Transitions in the State Machine

The state transitions shown in Figure 3 are exemplified below with the help of Figure 4(a). There are 3 Web services S_1, S_2 and S_3 with $RT_1 = 400ms, RT_2 = 700ms$ and $RT_3 = 900ms$, for simplicity, score decay for all Web services is kept linear.

Ready to Finish: If a Web service has been completely exhausted, i.e. all the data from it has been retrieved then its state is changed from *Ready* to $Finish$. A

Web service can be put to *Finish* state only when it is in *Ready* state and makes a data extraction. Figure 4(a) shows that after 2800ms, S_2 is put from *Ready* to *Finish* state. **Ready to Stop and Wait to Stop:** If a Web service is in *Ready* or *Wait* states then it should be put into *Stop* state if the following condition holds: if S_{result} already holds K join results, then the algorithm compares the local threshold τ_i with $\sigma(t_{result}^{(K)})$, the score of $K - th$ join result in S_{result}. If τ_i is less than or equal to it then it assigns *Stop* state to S_i. This essentially means that further extraction of data from this Web service will not produce any join result whose score is greater than the join results already in S_{result}. Figure 4(a) shows that after 2100ms the Web service S_3 is put from *Wait* to *Stop* state as its τ_3 is lower than $\sigma(t_{result}^{(K)})$. Whereas, S_1 is put from *Ready* to *Stop* state at 2500ms.

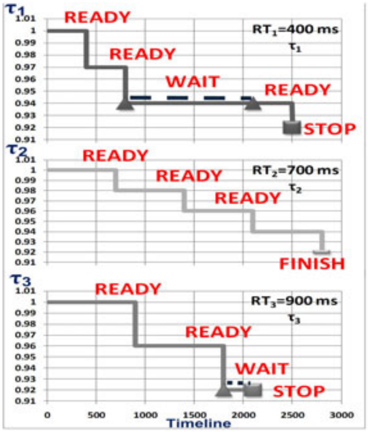
Ready to Ready, Ready to Wait, Wait to Ready and Wait to Wait: A Web service in *Ready* state is put to *Wait* state, or a Web service in *Wait* state is put to *Ready* state by analyzing the local thresholds of all other Web services which are in *Ready* state. Figure 4(b) presents the algorithm for *setState* function. Below is the explanation of the algorithm for a Web service S_i:

- Consider a set J containing all the Web services having local thresholds greater than that of τ_i and are in *Ready* state. The algorithm estimates the *time to reach* (ttr_j), for all Web services $S_j \in J$ to bring τ_j lower than τ_i as explained in Section 3.1.2.
- Thus, ttr_j is computed for all Web services in J and the maximum of these values is retained as ttr.
- If S_i is in *Ready* or *Wait* state and ttr is greater than or equal to RT_i then S_i is assigned *Wait* state, otherwise, it is put to *Ready* state.

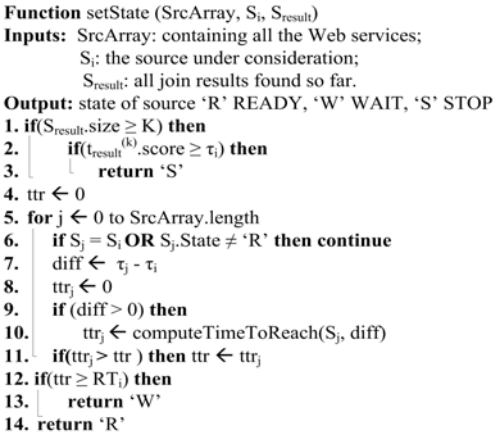
Figure 4(a) shows that, after 800ms, S_1 is put from *Ready* to *Wait* state because of bootstrapping phase, as no more than 2 data extraction calls are allowed during this phase from any Web service. This is explained below in this section. However, even after finishing the bootstrapping, at 900ms, it remains in *Wait* state as ttr_2 is 1900ms which is greater than RT_1. S_1 continues to be in *Wait* state at 1400ms and at 1800ms, as ttr is greater than RT_1. Similarly, at 1800ms, S_3 is put to *Wait* state from *Ready* state, as ttr_2 is 1700ms.

After 2100ms S_1 is put from *Wait* to *Ready* state as at this time ttr_2 is 0. Therefore, we need to resume data extraction from S_1 as well. Lastly, S_2 remains in *Ready* state during all the state transitions, till it moves from *Ready* to *Finish* state at 2800ms because it remains the Web service with highest local threshold i.e $\tau_2 = \tau$.

Bootstrapping: At the beginning data is extracted from all Web services in parallel. The phase before extraction of at least one chunk from all Web services is considered as bootstrapping phase. The Web services with smaller response time may fetch too much data in this phase. So, during bootstrapping, we limit maximum two data fetches from a particular Web service. The rationale is that these Web services have much shorter response time so they can catch



(a)



```
Function setState (SrcArray, S_i, S_result)
Inputs: SrcArray: containing all the Web services;
        S_i: the source under consideration;
        S_result: all join results found so far.
Output: state of source 'R' READY, 'W' WAIT, 'S' STOP
if($S_{result}$.size ≥ K) then
    if($t_{result}^{(k)}$.score ≥ $τ_i$) then
        return 'S'
ttr ← 0
for j ← 0 to SrcArray.length
    if $S_j = S_i$ OR $S_j$.State ≠ 'R' then continue
    diff ← $τ_j - τ_i$
    $ttr_j$ ← 0
    if (diff > 0) then
        $ttr_j$ ← computeTimeToReach($S_j$, diff)
    if($ttr_j$ > ttr ) then ttr ← $ttr_j$
if(ttr ≥ $RT_i$) then
    return 'W'
return 'R'
```

(b)

Fig. 4. (a) Execution of cPRJ with 3 Web services, over timeline against local thresholds. (b) The setState algorithm

up the other Web services with higher response times. It can be observed in Figure 4(a) that S_1 is put to $Wait$ state after making two fetches, at 800ms. Similarly, at 400ms S_1, and at 700ms S_2, are allowed to perform second fetch. The bootstrapping phase ends after 900ms.

Adaptivity to the Change in RT: Sometimes it is possible that a Web service S_i does not demonstrate the same response time as anticipated. To determine this, the proposed algorithm always computes the response time for every chunk and computes average of the last 3 observed response times. If the deviation is within 10% of the existing response time value, then the latter is retained. Otherwise, RT_i is assigned the average of its last 3 observed RT values.

4 Concurrent Pre-fetching with cPRJ

It is possible to profile a Web service S_i and identify if more than one concurrent calls can be issued to it. In such cases, instead of fetching one chunk at a time from S_i, the algorithm might issue $S_{i(conc)}$ concurrent calls. This helps in speeding up data fetching even further, as it acquires data from $S_{i(conc)}$ chunks in RT_i, the same time in which the baseline cPRJ gets one chunk.

This also requires modifications in the *setState* function while calculating the ttr, by incorporating the number of concurrent chunks extracted by S_i. Also, while issuing the data extraction calls, the algorithm has to check the number of chunks a Web service needs to bring its local threshold down to $\sigma(t_{result}^{(K)})$. If they are greater than or equal to $S_{i(conc)}$ then all concurrent data extraction calls can be issued. Otherwise, the number of calls is that suggested by the calculation.

This variant certainly reduces the time to find the top-K join as compared to the baseline version of cPRJ. However, it may incur some additional I/O cost because of concurrent data extraction.

Concurrent accesses to a Web service might also be considered an ethical issue as it prevents the other users from accessing the same service at the same time, especially in peak hours. However, in our case the total number of calls to a Web service will still remain almost the same even if we issue them concurrently. Secondly, the number of concurrent calls, in general is not high, and it should be issued only for the Web services with larger response times, or which exhibit a very low decay in their scores. As an example, in case of extracting data from the Web services `venere.com` and `eatinparis.com`, shown in Table 1, it will be useful to extract the concurrent chunks from them according to the ratio between their response times, provided their score decay per chunk is observed to be in the same ratio.

5 Experimental Study and Discussion

5.1 Methodology

Data Sets: We have conducted the experiments on both synthetic data, and real Web services. The experiments are based on the query in Example 1 by generating many different synthetic data sources with various parameter settings. The relevant parameters are presented in Table 2. The real Web services used for the experiments are presented in Table 1. These real services were queried for finding the best combination of hotels and restaurants in a city, for many different cities. For each city, we find the best combination of hotels and restaurants located in the same zip code. In order to consider more than two Web services, we have also extracted information about museums and parks from the real Web services. The experiments with synthetic data are performed with diverse and homogeneous settings of values for the parameters in Table 2. Homogeneous settings help us understanding the behaviour of individual parameter whereas, diverse settings help us simulating the real environment Web services, as we have observed that most of them have diverse parameter settings. For fairness, we compute these metrics over 10 different data sets and report the average. The experiments with the real Web services are conducted by fetching the data from real Web services for 5 different cities and the averaged results are reported.

Table 1. Real Web services used for experiments

	Web Services	Type of Information	Response Time	Chunk Size
1	www.venere.com	Hotels	900 ms	15
2	www.eatinparis.com	Restaurant (only for Paris)	350 ms	6
3	Yahoo! Local	Hotels, Restaurants, Museums, Parks	800-1200 ms	10
4	www.yelp.com	Hotels, Restaurants, Museums, Parks	900-1100 ms	10

Table 2. Operating Parameters (defaults in bold)

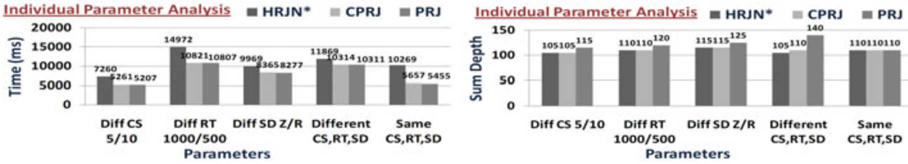
Full Name	Parameter	Tested Values
Number of results	K	1,**20**,50,100
Join Selectivity	JS	0.005, 0.01, **0.015**, 0.02
Score Distribution	SD	Uniform Distrib., **Zipfian Distrib.**, Linear Distrib., Mixed
Response Time	RT	**500/500**, 500/1000, 500/1500
Chunk Size	CS	**5/5**, 5/10, 5/15
Number of relations	m	**2**,3,4

Approaches: We compare three algorithms, HRJN*, PRJ and the proposed cPRJ while using *tight* bounding scheme. An important consideration is that HRJN* augmented with *tight* bounding cannot be beaten in terms of I/O cost, whereas PRJ cannot be out-performed in terms of time taken, provided the time taken for joining the data is negligible. Therefore, the proposed algorithm, cPRJ carves out a solution that deals in the trade off between I/O cost and time taken. Indeed, the parallel approaches should be efficient in terms of time taken than the serial data accessing HRJN* approach yet, the purpose of including HRJN* in the comparison is to elaborate the gain in terms of I/O cost when using cPRJ instead of PRJ.

Evaluation Metrics: The major objective of the proposed approach is to reduce the time taken to get the top-K results by minimizing the data acquisition time with the help of parallelism. So, we consider *time taken* as the primary metric for comparing different algorithms. This is the wall clock time, that is, starting from the first fetch till the $K-th$ join result is reported. The reduction in time is obtained by compromising on possibly some extra data extraction as compared to HRJN*. Therefore, we consider *sum depths* [5], total number of tuples retrieved from all Web services, as other metric for comparing the different algorithms.

5.2 Results

Experiments with Synthetic Data: In Figure 5 we show the results of the experiments for CS, RT and SD parameters while joining two Web services. In case of the homogeneous setting of the parameters, i.e. keeping all the parameters to the default values and setting different values for one of the three above mentioned parameters. This results into termination of data extraction from $(m-1)$, in this case, one data source earlier than the other data source, as explained in section 3.1. The proposed cPRJ algorithm is also based on these three parameters. Figure 5(b) shows that cPRJ incurs 1% more and PRJ incurs 8% more I/O cost than HRJN* in case of different CS values. For different values of RT and SD both HRJN* and cPRJ take the same I/O cost, and PRJ takes 8% more and 10% more I/O cost than HRJN* for different values of RT and SD, respectively. If we augment all these in one scenario then cPRJ incurs 3% more I/O cost than HRJN* and PRJ costs 29% more I/O cost than HRJN*.



(a) (b)

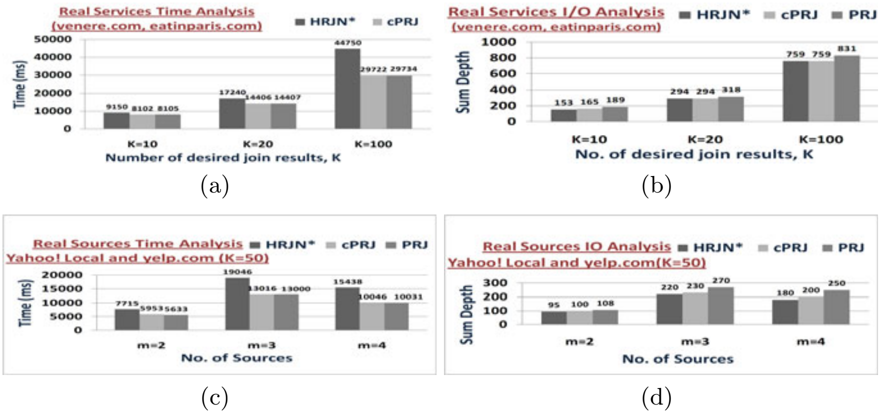
Fig. 5. Performance comparison of the algorithms on synthetic data sources for the parameters shown in Table 2

Whereas, Figure 5(a) shows that for all cases the time taken by both parallel approaches is almost same and is much lower than HRJN*. However, if CS, RT and SD are identical for all data sources, then all three approaches have almost same I/O cost and both parallel approaches take same time.

The overall performance of cPRJ is much better than PRJ in case of diverse parameter settings, as it has almost same I/O cost as of HRJN* whereas, it takes almost same time as of PRJ, whereas, PRJ has higher I/O cost than HRJN*. Thus, in the diverse settings it brings the best of both worlds.

Real Web Services: The experiments with the real Web services, which in general, have diverse parameter settings, confirm the same observations made on synthetic data, i.e. overall cPRJ performs much better than PRJ. We performed experiments for the query in Example 1 while interacting with the real Web services to get top-K join results. We have used different Web services, presented in Table 1. Figure 6(a) shows that both parallel approaches take same amount of time which is 20-25% less than HRJN*. The difference in time increase by increasing K. Figure 6(b) shows that the I/O cost incurred by proposed cPRJ is 5% more than ideal HRJN*, whereas, PRJ takes 8-10% extra data fetches. We have also performed experiments by varying the number of Web services involved in the search query. We add data for museums as third and data for parks as fourth Web service in our search. We use `Yahoo! Local` and `yelp.com` to fetch data for museums and parks. The results shown in Figure 6(c) show that both parallel approaches take almost same time and this time is 14-35% less than HRJN*. The difference in time taken by parallel approaches and HRJN* increases by adding more data sources, i.e., by increasing the value of m. The results presented in 6(d) demonstrate that cPRJ takes 4-11% more I/O cost than HRJN*, whereas, PRJ takes 13-38% more I/O cost than HRJN*.

The experimental results also show that other three parameters JS, m and K do not have any impact *alone*. They cannot be responsible for the early termination of a *single* data source. However, if SD, RT and CS have heterogeneous values, and if the overall impact of these values is that they enforce one or more data sources to terminate earlier than the others while using the parallel approaches, then JS, m and K also come into play. The results shown in Figures 6(a) and 6(b) show the role of K and Figures 6(c) and 6(d) show the behaviour of number of data sources m, involved in a query.



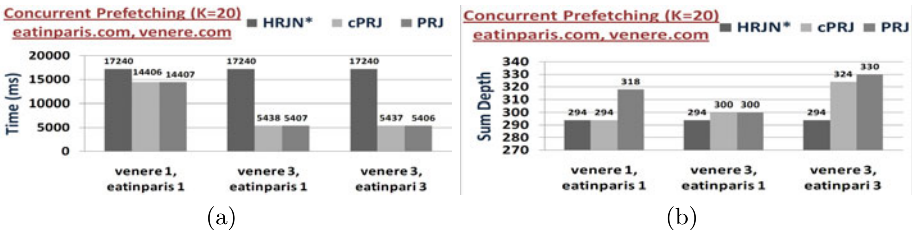
(a) (b)

(c) (d)

Fig. 6. Performance of the algorithms with real services. Figures (a) and (b) are for the experiments with `venere.com` and `eatinparis.com`. Figures (c) and (d) are experiments with different number of sources using `Yahoo! Local` and `yelp.com`

The method used to compute *ttr* is supposed to provide accurate estimates when the score decay is smooth. When this is not the case (e.g. when ranking of hotels is induced by the number of stars), it tends to underestimate the score decay. If it underestimates the score decay then the state machine may pause a Web service unnecessarily, which may increase the overall time. Conversely, in case of overestimation of the score decay, the state machine may not pause a Web service at right time, hence, it may incur extra I/O cost.

Concurrent Pre-fetching: The results in Figure 7 are based on an experiment which issues different number of concurrent calls to the real Web services, `venere.com` having response time 900ms and `eatinparis.com` having response time 350ms. We issue concurrent calls in two ways, firstly, based on the ratio between the response times of the two sources, and secondly, we issue three concurrent calls for both data sources without any consideration. The results show that in both cases the time decreases by almost 62% of the baseline cPRJ approach. This implies that `venere.com` takes most of the time to fetch the data to produce required number of join results, whereas, `eatinparis.com`



(a) (b)

Fig. 7. Figures (a) and (b) show the comparison of time and I/O for K=20, where cPRJ and PRJ perform different number of concurrent fetches on real Web services

takes one third or lesser time to fetch its data from the same purpose. Therefore, when we fetch three concurrent chunks from `venere.com` and one chunk from `eatinparis.com`, we get the best result. While observing the difference in the I/O cost, we find that first method of concurrent calls has proven to be almost as effective as baseline cPRJ whereas the second one has incurred 10% extra I/O cost than the baseline cPRJ. More than one concurrent data fetches from a Web service certainly minimize the time, however, using it in a *smarter* fashion can also help avoiding possible extra I/O cost.

6 Related Work

We are considering rank join operators with only sorted access to the data sources, therefore, we only discuss the existing solutions while respecting this constraint. The NRA algorithm [4] finds the top-K answers by exploiting only sorted accesses to the data. This algorithm may not report the exact object scores, as it finds the top-K results using bounds; score lower bound and score upper bound; computed over their exact scores.

Another example of no random access top-K algorithms is the J* algorithm [1]. It uses a priority queue containing partial and complete join results, sorted on the upper bounds of their aggregate scores. At each step, the algorithm tries to complete the join combination at the top of the queue selecting the next input stream to join with the partial join result and reports it as soon as it is completed. This algorithm is expensive in terms of memory and I/O costs as compared to HRJN* in most of the cases.

HRJN [7] is based on symmetrical hash join. The operator maintains a hash table for each relation involved in the join process, and a priority queue to buffer the join results in the order of their scores. The hash tables hold input tuples seen so far and are used to compute the valid join results. It also maintains a threshold τ and uses a data pulling strategy to compute join results. Some recent improvements in HRJN algorithm are presented in [5] and [12]. These algorithms use *tight bound* to compute top-K join results and show their comparative analysis.

Another interesting and objectively similar work has been done in [10], but the proposed algorithm *Upper* incorporates both serial and random accesses to the data sources, whereas, in our case we only use sorted access to the data sources. The commonality between the two approaches is that both cPRJ and *Upper* minimize the data extraction time by issuing concurrent data extraction calls and also exploit the pre-fetching of data while respecting the number of maximum concurrent fetches to the data sources.

7 Conclusion

We have proposed a new rank join algorithm cPRJ, for multi-way rank join while using parallel data access. This algorithm is specifically designed for distributed data sources which have a non-negligible response time e.g. the Web services available on the Internet. It uses a score guided data pulling strategy which helps computing the top-K join results. The results based on the experiments conducted on synthetic and real Web services show that the I/O cost of the proposed approach is nearly as low as optimal I/O cost of HRJN*, and it computes the join results as quick as PRJ approach which cannot be beaten in terms of time taken. cPRJ exhibits its strengths when the Web services have such diverse parameter settings which enforce one or more data sources to terminate earlier than any other data source while accessing them in parallel. We have also exploited the concurrent data fetching property of the Web services in order to get the data in even quick time. This reduces the time to compute the joins even further, but at higher I/O cost than baseline cPRJ. As a next step, we anticipate that this parallel rank join operator can be enhanced for pipe joins.

Acknowledgments

This research is part of the "Search Computing" project, funded by the European Research Council, under the 2008 Call for "IDEAS Advanced Grants".

References

1. Nastev, A., Chang, Y., Smith, J.R., Li, C., Vittor, J.S.: Supporting incremental join queries on ranked inputs. In: VLDB Conference
2. Brockwell, P.J.: Encyclopedia of Quantitative Finance (2010)
3. Ceri, S., Brambilla, M. (eds.): Search Computing II. LNCS, vol. 6585. Springer, Heidelberg (2011)
4. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. Journal of Computer and System Sciences 66(4), 614–656 (2003)
5. Finger, J., Polyzotis, N.: Robust and efficient algorithms for rank join evaluation. In: SIGMOD Conference, pp. 415–428 (2009)
6. Guntzer, U., Balke, W., Kiessling, W.: Towards efficient multi-feature queries in heterogeneous environments. In: International Conference on Information Technology: Coding and Computing, Proceedings, pp. 622–628 (2001)
7. Ilyas, I., Aref, W., Elmagarmid, A.: Supporting top-k join queries in relational databases. The VLDB Journal 13(3), 207–221 (2004)
8. Ilyas, I., Beskales, G., Soliman, M.: A survey of top-k query processing techniques in relational database systems. ACM Computing Surveys 40(4), 1 (2008)
9. Mamoulis, N., Theodoridis, Y., Papadias, D.: Spatial joins: Algorithms, cost models and optimization techniques. In: Spatial Databases, pp. 155–184 (2005)
10. Marian, A., Bruno, N., Gravano, L.: Evaluating top- queries over web-accessible databases. ACM Trans. Database Syst. 29(2), 319–362 (2004)
11. Martinenghi, D., Tagliasacchi, M.: Proximity rank join. In: PVLDB, vol. 3(1), pp. 352–363 (2010)
12. Schnaitter, K., Polyzotis, N.: Optimal algorithms for evaluating rank joins in database systems. ACM Trans. Database Syst. 35(1) (2010)