Politecnico di Milano
Dipartimento di Elettronica e Informazione
RESEARCH DOCTORAL PROGRAM IN INFORMATION TECHNOLOGY

Rank Joins for Web Based Data Sources

Doctoral Dissertation of:
Adnan Abid

Advisor:
Prof. Stefano Ceri
Tutor:
Prof. Barbara Pernici
The Chair of the Doctoral Program:
Prof.

2011 – XXIV

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 I-20133 — Milano

To my whole family.

Acknowledgements

Adnan

Milano,
January 2012

Abstract

Abstract in english

Contents

List of Figures

List of Tables

1 Introduction

Write breifly about
evolution of Web Search.
need and importance of ranking in Web search and databases

1.1 Motivation

Briefly discuss the Web Search solutions involving ranking..
Put forth the short comings in the existing solutions..
Main focus shall be that almost all the existing solutions focus on reducing the I/O cost which effectively reduces the overall time to fetch the data as well. However, they do not consider the cases when the data sources are remote and hence the time to process the data is negligible as compared to the time to acquire the data.
Apart from this, in threshold based rank join algorithms, it is commonly observed that we find a join result in the early stages of the process but we are unable to report it till the threshold allows us to do so. This invites us to come up with an approach to report the join results to the user with certain probabilistic guarantees.

1.2 Contribution of this Thesis

contribution This thesis provides time and I/O efficient rank join algorithms for the data sources which have a non negligible time associated for the data acquisition, and furthermore these data source suffice the requirements of applying rank join algorithms.

It provides efficient rank join strategies for different join topologies e.g. Multi-way Rank Join and Pipe Rank Join.
It also presents an approach to provisionally report the join results obtained in a rank join, to the user, with certain probabilistic guarantees. This further helps reducing the time to produce the join results, while compromising on the quality of the join results as compared to the deterministic approaches.
We also present the usefulness of this work while using these rank join algorithms in an application. We choose Search Computing application for this purpose and see how our defined algorithms can be utilized in different components in different settings.
We also analyze the effectiveness of our proposed algorithms over the existing algorithms while testing them in various different parametric settings and environments. This helps us in understanding the main operating parameters which affect the performance of the proposed algorithms.

1.3 Thesis Overview

The next chapter introduces the rank joins and we discuss the major rank join algorithms which are related to our context. We also introduce the Web search and its evolution, where we emphasize on the need of ranking in the Web search applications.
In chapter 3, we present two main rank join topologies for computing the top-K joins efficiently in the context of Web based data sources e.g. Web services.
Chapter 4, provides a probabilistic technique to report the join results to the user with certain confidence.
In chapter 5, we investigate the ways in which we can integrate our proposed rank join techniques with in a Web search application. We use Search Computing application for this purpose. This helps us in assessing the effectiveness of our approach with in real Web searching systems.
Lastly, chapter 6 provides the final discussion and future outlook of the work done in this thesis.

2 Ranking and Web Search

2.1 Rank Joins or Top-K Queries in Databases

The information systems process data in different ways in order to produce and rank the query answers. In many application domains, end-users are mainly interested in getting most relevant query answers, instead of exploring large number of weakly related answers. These queries are known as top-K queries, as the user is interested in getting 'K' best answers. Different emerging applications warrant the support of top-K queries in an efficient manner. For instance, many applications in the context of information retrieval [Salton and McGill 1983] involve top-K queries; similarly, in these types of queries are desired in the domain of data mining [Getoor and Diehl 2005]; furthermore, in the Web search applications, the performance and effectiveness of meta-search engines heavily depends upon the ways in which they process and combine rankings from different search engines. Above all, most of these applications process the queries which involve joining and aggregating multiple inputs to compute top-K results which are most relevant to the user's query.

A common and simple way to compute the top-K objects or tuples is to assign scores to all objects using some scoring function. The score of an object demonstrates its significance according to its properties (e.g., rent and area of an apartment object in a real estate database, or color and texture of an image in a multimedia database). The objects are evaluated and ranked by computing their total scores based on multiple scoring predicates. Furthermore, top-K processing is involved in various areas of database research which include query optimization, indexing methods, and query languages. Therefore, the impact of efficient top-K processing is becoming imperative in an various data processing applications. We present some example scenarios from the real-world where efficient processing of top-K queries is required. These examples also highlight the importance of adopting efficient top-k processing techniques in traditional database environments.

Example 2.1: Consider a user is interested in finding 5 least expensive places in a city where the combined cost of renting an apartment and tuition fee for school for a year is minimum. Let us assume there there are two data sources, *Apartments* and *Schools* from where we can get the information about apartments and schools, respectively. The *Apartments* data source provides an ordered list of apartments based on their respective rents and their locations. Whereas, *Schools* data source provides an ordered list of schools based on the tuition fee.

A naïve way to answer the query presented in Example 2.1 is to retrieve all apartments from the data source *Apartments* and schools from the data source *Schools*. Then join the objects from both lists which are located in the same location. Compute the total expenses for each join result by adding the rent and school fee together. Now, the five cheapest pairs (join results) constitute the final answer to this query. The important consideration is that unless we process the data from both data sources completely, the top five results cannot be returned to the user. So, for large numbers of apartments and schools, such a query involves a lot of processing while using this traditional method, since it requires expensive join and sort operations for large amounts of data.

Example 2.2: A top-K query from a video system database is that a user wants to find 10 most similar video frames to a given image with respect to some visual features, e.g. on the basis of colour and texture similarity. Consider a database of video system which stores many visual features which are extracted from each video object (frame or segment). These features may include color histograms, color layout, texture, and edge orientation. These features are stored in separate relations and are indexed using high-dimensional indexes which support similarity queries.

The above mentioned query highlights the importance of efficient processing for top-K similarity queries. Here the user can provide a function that combines similarity scores in given features to formu-

late an overall similarity score. For example, the overall similarity score of a frame f with respect to a query image q can be computed using a given score aggregation function, such as,
0.5 ×ColorSimilarity(f , q)+0.5× TextureSimilarity (f , q).

Again, a simple way to address such a multi-feature query is by sequentially scanning all database objects, while computing the score of each object according to the features under consideration, and computing the total score for each object by using the score aggregation function. However, this approach suffers from scalability problems with respect to database size and the number of features. As an alternative, we can map the query into a join query that joins the output of multiple single-feature queries, and then sorts the joined results based on combined score. But, this approach also does not scale with respect to both number of features and database size since all join results have to be computed then sorted.

The main problem with sort-based approaches is that sorting is a blocking operation which requires computation of all the join results. Although the input to the join operation is sorted on individual features, yet this order is not exploited by conventional join algorithms. Hence, sorting the join results becomes necessary to produce the top-K answers. Therefore, it requires embedding rank-awareness in query processing techniques to provide a more efficient and scalable solution.

Figure 2.1 shows alternative plans for processing the query to get top-K join results from two data sources A and B. One plan uses nested loop join to compute the joins and then uses sorting to extract top-K join results. The other plan sorts the base relations and then uses rank join which incorporates ranking in the join operation, to produce top-K join results. At the same time, below each of the plans we show the amount of data that needs to be processed in order to get the top-K join results while using the respective plan. We also see that in the case of nested loop join complete data needs to

be processed, whereas, the rank join exploits the sorting to produce the desired number of top-K join results while accessing a subset of the whole data.

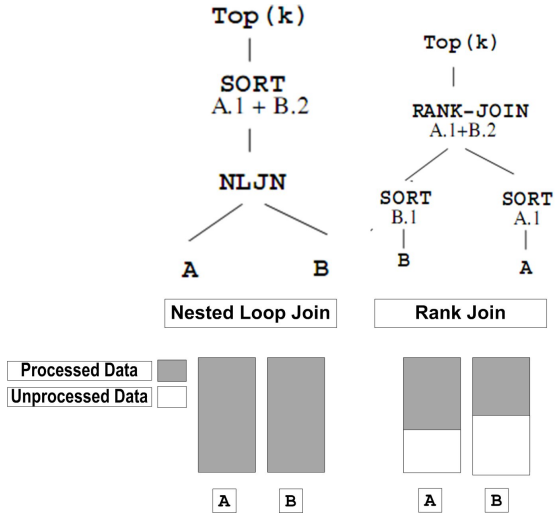


Figure 2.1: Rank join vs nested loop join.

2.2 Taxonomy of Top-K Queries

In this section, we discuss the state-of-the-art top-k query processing techniques in relational database systems. We give a detailed coverage for most of the recently presented techniques focusing primarily on their integration into relational database environments. We also introduce a taxonomy to classify top-k query processing techniques based on multiple design dimensions, described in the following:

Query Model Dimension

Firstly, we present the classification of top-k query processing techniques based on the query model they assume. There are three

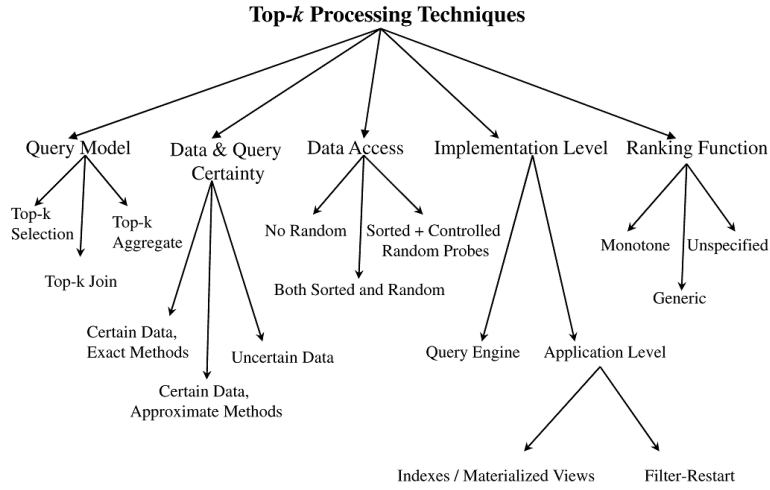


Figure 2.2: Taxonomy of Top-K Joins

different subcategories in the query model dimension:
i) Top-K Selection Query Model:
Some techniques assume a selection query model, where scores are attached directly to base tuples. As an example consider a user wants to find the top 5 images which are most similar to a given image based on certain features e.g. colour and texture. NRA algorithm [6] presented by Fagin et. al. is an example of this type of techniques. We briefly discuss it in Section 2.3.
ii) Top-K Join Query Model:
This branch of the taxonomy involves the rank join techniques which assume a query model, where scores are computed over join results based on some scoring function. As an example, finding five places in a city based on lowest tuition fee and prices of the apartments, requires joining two different data sources of Apartments and Schools. There are many top-K join techniques which fall in this category e.g. NRA-RJ [] by Ilyas works well when the join condition is equi-join. Other techniques under this category are J^* algorithm [1] by Nastev

et. al, Rank Join [10] by Ilyas et. al, and PREFER []. We discuss Rank Join by Ilyas et. al. in Section 2.3.
iii) Top-K Aggregation Query Model:
As the names suggests in this third query model category the scores are computed based on groups of tuples. As an example, we may find top 10 employees averaging on their age and salary combined. Li et. al [] (2006) have presented an algorithm for dealing with the challenges in aggregation based rank join algorithms.

Data Access Dimension
The data access dimension classifies the top-K join queries is based on ways in which we can access the data from the underlying data sources. The main data access techniques involve the availability of: only sorted access; both sorted and random accesses and lastly, sorted access with controlled random access. Here we need to understand the meaning of sorted and random accesses. A sorted access is the one which accesses the ranked list in a sequential order, i.e. a high scoring object has to be traversed before the low scoring object. On the other hand if the score of an object is required directly without traversing the objects with higher or lower scores then we call it random or direct access. Based on these access methods rank joins can be categorized into three sub categories. *i*) Both Sorted and Random Access:
This category ensures that all the data sources involved in a query offer both methods to access the data objects. The examples of such algorithms are Threshold Algorithm (TA) by Fagin et. al. [6] and Quick Combined algorithm [9] by Guntzer et. al. We have discussed the TA algorithm briefly in Section 2.3.
ii) No Random Access:
In this category, as the name suggests the data can only be accessed in sequential order from all the data sources. No Random Access algorithm (NRA) [6] by Fagin et. al. and Stream Combine algorithm [8] by Guntzer et. al. are the two such algorithms in the literature. We discuss NRA algorithm in Section 2.3.
iii) Sorted Access with Controlled Random Probes:

In this category the top-K queries involve at least one data source which offers sorted access, though it may offer random access as well. Here the assumption is based on practical observations that random access is expensive as compared to the sorted access. Therefore, in the approaches which fall under this category try to reduce the number of random probes in order to produce top-K join results. Examples of such algorithms are Rank Join algorithm [] by Ilyas et. al., MPro algorithm by Chang et. al. and Upper and Pick algorithms [] [] by Bruno et. al. We have discussed few of them in the coming section.

Implementation Level Dimension

In this category of the taxonomy we present as to how the top-K join algorithm is going to be integrated with the system. There are two sub categories for this purpose. One is application level and the other is query engine level.

i)Application Level:

One way is to embed the top-K join algorithm at the top of the query engine. This way the query engine works in its own way and the rank join algorithm can leverage from internal database objects like indexes and materialization etc. to perform more efficiently. Furthermore, new data access methods and specialized data structures can also be used. However, the main processing for the top-K queries is conducted outside the query engine. The examples of such algorithms include algorithms presented by Chang et. al. [], and Hristidis et. al. [].

ii)Query Engine Level:

The algorithms of this category involve changes at the database engine level, which helps executing the query in a rank join perspective. Thus, ranking plays a role in the optimization and processing of the query. Some of such techniques have introduced new join operators which support ranking and are called rank join operators. As an example, in rank join algorithm [] presented by Ilyas et. al. introduces a rank join operators. Whereas, following the same lines Li et. al. [] have presented the extensions to the existing query algebra for incorporating rank awareness.

Query and Data Uncertainty Dimension

This category depends upon the query environment and nature of data involved in the query processing. As an example, in data warehouse environment we have to process a huge amount of data in order to get exact query answer. So, in such applications we may sacrifice the accuracy of query answers in order to scale up the performance of the system. Therefore, we may report approximate answers for top-K queries. On the other hand, the uncertainty may appear in the data itself, e.g. the data collected from streams and sensor networks is not accurate data and it needs to be cleaned. Therefore, in such cases the queries are formulated and processed while taking the data uncertainly into consideration. From the top-K queries we have defined the following sub-categories from the query and data uncertainty dimension:

i) Exact Methods over Certain Data:

Here we have exact data and we process so as to get the answers with deterministic guarantees. Most of the rank join algorithms fall in this category, where deterministic data is processed by deterministic algorithms.

ii) Approximate Methods over Certain Data:

This sub-category involves the processing of top-K queries over deterministic data, but here the algorithms use approximations to produce the nearly optimal results and this way they compute the results in an efficient way. These kind of algorithms are generally used in Decision Supports Systems and data warehouses. These approximate answers are generally associated with probabilistic guarantees. The examples of such algorithms are Theobald et. al. [] and Amato et. al. []. We have discussed such algorithms in Chapter 4 where we present our algorithm which works on certain data and produces top-K join results with probabilistic guarantees.

iii) Uncertain Data:

This sub-category involves query processing of the data which is obtained from streams or sensor networks, i.e. the data needs to be cleaned before it is processed. Some algorithms which belong to this category use probability as the only score model, whereas,

others methods exploit both score and probability dimensions. The examples of such algorithms are Re et. al [] and Solaiman et. al []. The work presented in this thesis is based on deterministic data therefore we do not discuss any such algorithms further.

Ranking Functions

The rank joins can be further classified based on the type of ranking or scoring function. There is a vital property of rank joins which is to define the upper bound on the object's scores. These ranking or scoring functions help computing these values. We classify the rank joins based on the ranking functions in the following sub-categories:

i) Monotone Ranking Function:

A monotone ranking function easily computes upper bounds for the objects. A function F, which is defined over predicates $p_1, ..., p_n$, is a monotone scoring function if $F(p_1, ..., p_n)$ le $F(q_1, ..., q_n)$ whenever p_i le q_i for every i. Most of the rank join algorithms involve monotonic scoring functions as they widely cover many practical scenarios and are enriched with the efficient processing abilities. As an example the Threshold Algorithm (TA) [6] by Fagin et. al. uses monotone scoring function. All the algorithms proposed as a contribution for this thesis involve monotone ranking functions.

ii) Generic Ranking Function:

Some times ranking functions are expressed in the form of numerical expressions. In order to process such ranking functions efficiently, numerical optimizations and indexes are used. As an example, Zhang et. al. [] address the issues of top-K query processing for such generic ranking functions.

iii) No Ranking Function:

In this category we can see rank query processing which is conducted without any ranking function. As an example, skyline queries are processed without a ranking function. These queries produce the best objects which cannot be dominated by other objects based on certain attributes. Borzsonyi et. al. [] and Yuan et. al. [] present such algorithms. However, in the context of our thesis we do not discuss such algorithms.

2.3 State of the Art Rank Join Algorithms

In this Section we present some classical rank join algorithms which are the basis for many recent rank join algorithms. Firstly, we will present Threshold Algorithm [6] which was presented by Fagin. Then we present a variant of threshold algorithm which only involves sorted data access and is called No Random Access (NRA) algorithm [6]. Thirdly, we present another algorithm HRJN [10] which only allows sorted data access to the data sources. Lastly, we present [13] Upper and Pick [**?**] algorithms which present relatively similar work to our approach. All these approaches use monotonic scoring function, and the query model is top-K join query model, and in terms of data they deal with certain data, and can be implemented at any level. However, in terms of data access some assume both sorted and random access methods, whereas others assume only sorted access method.

Threshold Algorithm

Threshold algorithm (TA) [] was presented by Fagin et. al. in 2001. It assumes that the data sources exhibit both sorted and random data accesses. This algorithm scans multiple lists comprising of different rankings of same data objects. An upper bound τ is maintained to compute the maximum score of unseen objects. The score function computes the upper bound using the scores of the last seen objects in each list. This values of upper bound is updated every time a new object is observed from any of the lists. As soon as an object is observed in one list, the algorithms looks up this object in the rest of the lists and then computes the score of this object using the score function. An object is reported to the user if its score is greater than the upper bound τ. Figure 3.1 illustrates the processing of TA.

Consider two data sources L_1 and L_2 which contain same set of objects with two different scoring predicates. The range of scores for these objects in both data sources is [0,50]. Assume both sources

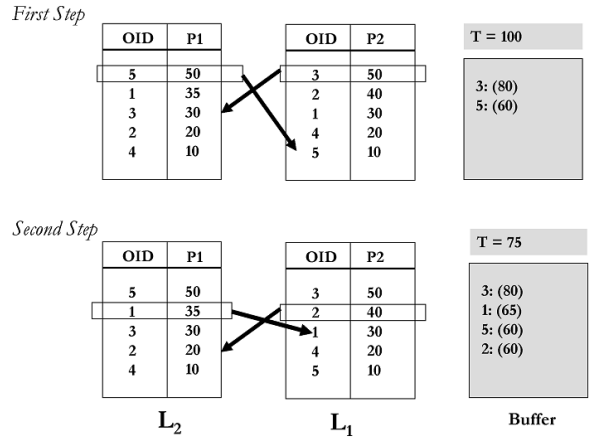


Figure 2.3: Threshold Algorithm (TA)

support sorted and random access to the data objects. Consider that simple linear addition is the score aggregation function. Figure 2.3 shows the execution of first two steps of TA. In the first step, the algorithm retrieves the top object from each data source, and then looks it up in the other data source. Then it uses the score aggregation function to compute the exact score of these objects. These seen objects are stored in a buffer in descending order of their exact scores. The upper bound or threshold value, τ, is computed by applying the score aggregation function to the last seen scores in both lists, which after the first step results in 50+50=100. Now, at this stage, both seen objects have scores less than τ, therefore, no results can be reported. After the the second step, the threshold τ drops to 75, which allows object 3 to be safely reported since its score is above τ. The algorithm continues like this until it manages to report K objects or the data from the sources is finished.

No Random Access Algorithm

NRA algorithm [] assumes that the data sources allow only sequen-

tial access to the data objects and random access is not supported. This algorithm may not report the exact scores of an object, since it computes the upper and lower bound of the score of an object. The lower bound score of an object t is computed by providing the score aggregation function the know score value for t and the least possible score value for t in the other data source. Whereas, the score upper bound of t is computed by providing the known score of t and the maximum possible score of t in the other data source, which is the same as the last seen scores in the corresponding data source. This allows the algorithm to report a top-k object even if its score is not precisely known. Specifically, if the score lower bound of an object t is not below the score upper bounds of all other objects (including unseen objects), then t can be safely reported as the next top-k object.

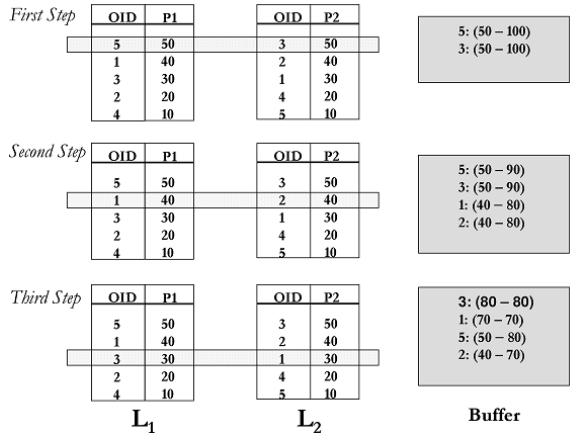


Figure 2.4: No Random Access Algorithm (NRA)

NRA Example: Consider two data sources L_1 and L_2, and each data source contains a different ranking of the same set of objects based on different scoring predicates. The scores of the objects in both data sources fall in the range [0, 50]. Consider both data sources only support sorted data access to the ranked data objects,

and the score aggregation function is simple linear addition. Figure 2.4 shows the first three steps of the NRA algorithm. In every step one object is retrieved from all data sources or lists. After the first step, the algorithm retrieves the first object from each data source. The algorithm then computes lower and upper bounds for the scores of the objects. For example, object 5 has a score range of [50, 100], as its known values is 50 and unknown value cannot exceed 50 and cannot be less than 0. The seen objects are stored in a buffer in the descending order of their score lower bounds. No object can be reported at this stage as the score lower bound of object 5, the top buffered object, does not exceed the score upper bounds of all other objects. Similarly, after the second step two more objects are added to the buffer, and score bounds are updated for score the other buffered objects. After the third step, the scores of objects 1 and 3 are completely known. Here, we can observe that the score lower bound of object 3 is greater than or equal to the score upper bound of any other object (including the unseen ones), so object 3 can be safely reported as the top-1 object. It is pertinent to note that at the same stage we cannot report object 1, becauses the score upper bound of object 5 is 80, which is larger than the score lower bound of object 1.

Hash Rank Join Algorithm

One example of rank-aware query operators that support pipelining is the Rank Join operator []. This algorithm, integrates the joining and ranking tasks in one efficient operator. There are commonalities between rank join and the NRA algorithm []. Both these algorithms perform sorted access to get tuples from each data source. There are two main differences between rank join and NRA algorithm. Firstly, NRA joins the lists of same objects, which is not the case in rank join. Therefore, unlike NRA where one object in one list has exactly one object in the other lists with which it can be joined, in case of rank join one object in one list can be joined with many others in the other list. That is why the rank join only stores the

completly seen join results in the buffer. Consequently, the Rank-Join algorithm provides exact scores of the join results, while the NRA algorithm reports bounds on scores. The other difference is that the NRA algorithm fetches one object from each list or data source in each iteration. The Rank-Join algorithm has adaptive strategy to extract object or tuples from a data source, and the algorithm adaptively chooses the data source from which more data needs to be fetched. The algorithm maintains a threshold τ which bounds the scores of the undiscovered join results. All join results with score greater than or equal to the threshold value are reported to the user. The algorithm continues to work unless K join results are reported. A two-way hash join implementation of the Rank-Join algorithm, which is called Hash Rank Join Operator (HRJN), was presented in Ilyas et al. []. HRJN is based on symmetrical hash join. This operator maintains a hash table to store and process the objects retrieved from each relation, and it maintains an output buffer as a priority queue to store the join results in the order of their scores.

Upper and Pick Algorithms

The Upper [**?**] and Pick [**?**] top-K algorithms assume that at least one of the data sources provides sorted access, while random accesses are scheduled to be performed only when needed. Both these algorithms are proposed in the context of Web-accessible sources. The main emphasis of these algorithms is that in the context of Web data sources random access is expensive as compared to the sorted access. Therefore, these algorithms minimize the number of random accesses in order to obtain top-K join results. The main purpose of having at least one data source with sorted-access is to obtain an initial set of candidate objects. These algorithms control the random accesses by selecting the best candidates, based on score upper bounds, to complete their scores.

The Upper algorithm, probes objects which have considerable

chances to be among the top-K objects. Firstly, a sorted access is made to get some candidate objects which are inserted into a priority queue based on their score upper bounds. After every data extraction from the sorted data source the score upper bound of unseen objects is updated. An object is reported and is removed from the queue if its score lower bound is higher than the score upper bound of any unseen object. The algorithm adaptively chooses the best source that should be probed next to obtain additional information for candidate objects.

In the Pick algorithm, the next data object to be probed into is chosen so as to minimizes a distance function, which is defined as the sum of the differences between the upper and lower bounds of all objects. The source to be probed next is selected at random from all sources that need to be probed to complete the score of the selected object.

Major Components of a Rank Join Algorithms

If we analyse the above mentioned rank join algorithms, the we can figure out that there are two main components of all rank join algorithms.
i) Bounding Scheme
ii) Data Pulling Strategy

We explain the main features of these components as follows:
Consider a query Q whose answer requires accessing the data from two tables S_1 and S_2 in a relational database. 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 both tables are sorted in the descending order of the score associated to them, where the score reflects the relevance with respect to the query. This sorting of data objects in the respective relations fulfils the qualification for the data so as to apply a rank join algorithm. Let t_i denote a tuple, $t_i^{(d)}$ tuple at depth d and $t_i^{(d+1)}$ tuple at depth $d+1$ for 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.

Bounding Schemes

Let $t = t_1 \bowtie t_2$ denote a join result formed by combining the tuples retrieved from two relations, where t_i is a tuple that belongs to the relation S_i. This join result is assigned an aggregated score based on a monotone score aggregation function, $\sigma(t) = f(\sigma(t_1), \sigma(t_2))$. Let τ denotes the threshold value, which is the maximum score of a join result that can be computed by joining the unseen tuples of the relations with the seen or unseen tuples of the rest of the tables. This calculation of threshold helps in formulating a bound. 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. The computation of threshold value depends upon the join predicate and data access methods. As an example, if the data from two relations is joined based on the tuple or object id and the data access method is sequential data access, then we can compute a upper and lower bound scores of each join result. We report that join results whose lower bound exceeds the upper bound of the rest of the join results, as discusssed in NRA algorithm above. Whereas, if we are allowed random data access in the same scenario, then the threshold is computed as a single score value, as we can observe in the TA algorithm discussed above. Similarly, if the objects or tuples in the tables are joined based on a join attribute value other than the object id, and data access method is sequential, then the threshold is computed as a number.

Data Pulling Strategy:

Data pulling strategy provides a mechanism to choose the *most suitable* data source to fetch the data [10]. Certainly, there is an objective behind the mechanism of the pulling strategy, e.g. *Hash Rank Join* (HRJN*) operator has the objective of optimizing the I/O cost i.e. total number of tuples to get the top-K join results. The data pulling strategy used in HRJN* chooses that relation for data extraction whose $\tau_i = \tau$, the ties are broken by total number of tuples which have been retrieved. Whereas, in case of Threshold Algorithm (TA), data pulling strategy is round robin strategy, which

extracts next sequential tuple from each relation in a round robin fashion, and then finds its corresponding matches from the other relations using random data access.

2.4 Adaptation of Rank Joins for Web Based Data Sources

The search systems over the internet have been evolving continuously. There are some generic search engines like Google, Yahoo and Bing etc. However, there exist a large number of domain specific web searching services which provide the user with the answers to certain specific questions: e.g. finding the movies by the IMDB and finding books by Amazon etc. These large number of web searching services have triggered the Web search community to design multi-domain searching systems while using the existing searching services. We can already see some examples of such systems e.g. Yahoo Pipes and Expedia. These systems entertain multipurpose user queries and serve the user requests by means of many existing services and respond back to the user by aggregating and ranking the results obtained by each service. A very important aspect of this search is that the user is interested in most relevant and appropriate results instead of large number of partially relevant results.

2.4.1 Classification of Web Services

We can categorize the Web services available on the Internet under different categories. These categories are described as follows:

Ranking Dimension

The Web services can be classified based on the way they order the data objects with their relevance to the query. They can be categorized into *Exact* and *Search* services []. An *Exact* Web service is the one which provides the results without computing any relevance of these objects to the given query. A *Search* Web service provides the output after scoring and sorting the resulting objects based on their relevance to the given query e.g. if we have a service for finding the hotels in a city, it may respond with the hotels sorted with its stars, or this ranking and sorting can be on the price as well, based on the given query.

Selectivity Dimension

The Web services can be categorized into *Selective* or *Proliferative* [], based on a factor called *selectivity*. The selectivity of a data source is the average number of tuples a service outputs in response to an invocation. So, if the maximum *selectivity* of a Web search service is 1 then it is *selective* and *proliferative*, otherwise.

Data Provision

The services can also be categorized based on how the data objects are reported in response to an invocation. The Web services can be categorized as *chunked* or *bulk* []. The former provides the data objects in pieces i.e. not all the results in one call and we have to call the Web service again to fetch the next set of results unless we are satisfied or the result set is finished. This phenomenon is generally known as *pagination*, where few data objects are reported on one page and then we have to request for the other page to get more data. Whereas, the latter approach gives all the results in one call. This phenomenon of chunking gives rise to some optimization dimensions as it allows to process the data before actually getting the complete data set in response to a query.

Apart from this, there are some general characteristics which are associated with the Web services, which are the following: in case of a *chunked* Web service, one *chunk* or *page* of information is provided in a specified amount of time which is generally called average response time of the Web service. Similarly, for a *chunked* Web service, the number of objects which are retrieved in each *chunk* or *page* are considered as *chunk size* for the Web service. Whereas, in case of a *bulk* Web service, there is an average response time associated with it, i.e. the average time that it takes to respond to a given query. There is not *chunk size* associated with the *bulk* Web services.

Data Access

Another very important consideration about these services is that there can be different data access methods for retrieving the information from the Web services, namely, *sorted* access, *random/direct* access or both *sorted* and *random* accesses. While using *sorted* ac-

cess we can only access the data in a sequence and cannot find any information randomly; in *random/direct* access we can find a certain record directly, e.g. by using the object Id. *Random* or *direct* access is rather expensive than *sorted* access. Note, that this is similar to the data access dimension discussed in Section **??**.
Another dimension to observe the data aquisition from the Web services is whether to invoke them in *parallel* or in a *pipe* line. This decision can be made by asssessing the precedence constraints enforced by the input/output bindings. Since, the overall results obtained by processing the data of Web services returns a single sequence of results, i.e. in both cases we say that the two services are joined. In case of parallel configuration data from the services is extracted simultaneously. Independent services can be invoked in parallel, and their results are processed as they are retrieved. In case of input/output dependencies, independent service is invoked first and then based on its output the dependent service is invoked. However, even when service calls have precedence dependencies, determined by the input/output bindings, it is not necessary to wait for the complete execution of the first service (in a blocking style), as its results can be fed as input to the second service as soon as they are available, so as to "pipeline" the join execution.

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 [5]. The Web services, e.g. `Yahoo! 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
```

In order to execute the above mentioned query let us assume that there are two Web services *hotel* and *restaurant*. These Web services can be invoked in parallel and we assume that there is no dependency for the invocation of these services. Figure 2.5 shows the execution plan for parallel join of this example query.

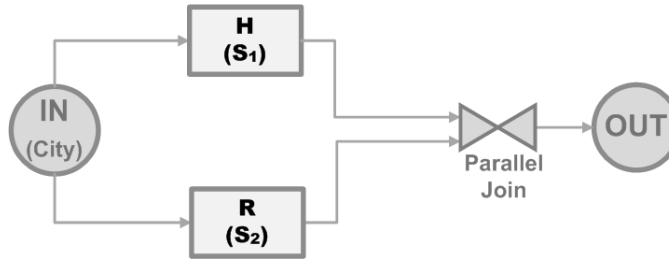


Figure 2.5: Example of parallel rank join.

Pipe rank joins extract data from data sources which may not be invoked for data extraction in parallel, e.g. the output of one data source is needed to invoke the other data source. Let us assume that for the example query in Section **??** the *restaurant* Web service provides the information about restaurants in a given city which are located in a given zip code. This exposition of restaurants allows to choose pipe join topology in which we extract data from the *hotel* Web service and then for all the objects observed from the *hotel* Web service we find the newly encountered zip code values. Then we pass on this zip code value to the *restaurant* Web service to extract the information about the restaurants located in the given zip code. Similarly, if *hotel* Web service also allows random access based on zip code information for a city, then we can swap the Web services in pipe line sequence. Figure 2.6 presents the execution possibilities of pipe rank join for the example query given in Section **??**.

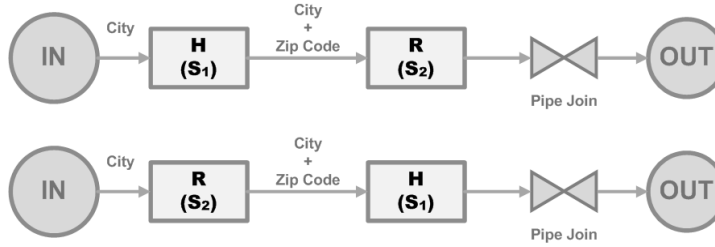


Figure 2.6: Possibilities for pipe rank join.

2.4.2 Hypothesis

There are existing approaches for parallel rank join topology [1][6][8][11][12]. Furthermore, recent solutions to rank join problem [7][10][15] 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. HRJN* [10] 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 [7] and [15].
Similarly, in [14] the authors provide a way of computing top-K join results using pipe join, with minimum I/O cost. We refer to it as *serial Pipe Rank Join* (sPRJ). Indeed, this optimization of the I/O cost in all these parallel and pipe join algorithms 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. In the next part of this chapter we present the parallel and pipe topologies for rank join algorithms which exploit parallel data access.

Apart from the above mentioned parallel data extraction methods to improve the query answering time in the context of query execution with Web services as data sources, we also address the issue of query processing time from another dimension. While analysing the state of the art rank join algorithms we observed that most of the time the real top-K join results are obtained much earlier as compared to the time when they are reported. The reason for this delay is the bounding scheme or the threshold value. As we discussed above, no join result can be reported unless its aggregated score is above or equal to the threshold value. So, in most of the cases, the join results are obtained and then the algorithms keep on extracting more data so as to bring the threshold value down and produce more join results. Finally, when the threshold falls below the score of an observed join results then it is reported to the user. This phenomenon triggers the need of reporting the already obtained join results with certain probabilistic guarantees with which they will appear among the top-K join results. This improvement is imperative in two cases, firstly, when the computation of top-K join involves huge amount of data and it is associated with a very high I/O and time costs. Secondly, when the data sources are Web services, which have non-negligible response time associated to them. In the latter case, the time to fetch the data is a bottle neck and it may delay the reporting of already observed join result which may be one among the final top-K join results. There are existing techniques for probabilistic measures for processing the top-K join results but most of them [] [] focus on discarding the partial join re-

sults which have a very low probability to be among the final top-K join results. We look forward to a probabilistic method which computes the probabilities of the currently observed top-K join results with which they may appear among the final top-K join results. A similar work has been conducted in [] and has been further improved in []. However, this computes the overall probability of having top-K join results at one particular stage. Another difference is that it is based on the TA and NRA algorithms. Whereas, we focus our work on rank join operator, which has different joining methodology as it does not join the objects based on object ids.

Therefore, we have two main research questions which we address in this thesis.

i) Exploit parallel data extraction for both parallel and pipe join topologies for processing the rank join algorithms in the context of Web based data sources e.g. Web services. The intuition is that the parallel data extraction surely reduces the time to compute the top-K join, however it may result into extra I/O cost. The objectives are to keep the I/O cost near to the optimal cost, while reducing the time to compute the top-K joins as much as possible.
ii) Minimize the time to compute the top-K join results by reporting the observed join results as the top-K join results with probabilistic guarantees instead of deterministic guarantees. Intuitively, we can say that this probabilistic reporting of the join results will help reducing the time to compute the join results as well as the I/O cost to compute the top join results. However, at the same time the quality of the results may suffer, as this method is prone to report the join results which actually are not top join results. Therefore, in this research question the objective is to avoid reporting these unwanted join results while using probabilistic guarantees.

3 Rank Joins for Web Data Sources

In this chapter we present the rank join algorithms for parallel and then pipe join topologies. We exploit the possibilities with which parallel data access can be used to improve the overall time to compute the top-K join results using these join topologies.

Firstly, we present a rank join algorithm which uses parallel join topology with two or more Web services. We assume that the Web services involved in these joins are equipped with only seqential data access and provide the results in descending order of the scores of their objects with respect to the given query. So these Web services are *search* Web services. We also assume that the data can be obtained from these Web services in the form of *chunks*, therefore, these are *chunked* Web services. Lastly, the selectivity of these Web services is higher and they are $proliferative$ Web services i.e. they provide multiple objects with each invocation. Above all, in the first part we present top-K join algorithm with parallel data access which involves two or more *search*,*chunked* and $proliferative$ Web services which are only equipped with *sorted* data access. We discuss different possible ways to use parallel data access and then see which approach is most suitable. We also present variants of this baseline approach which can be used under different situations and needs. Lastly, we analyse the proposed rank join algorithm with various operating parameters.

Secondly, we present a rank join algorithm which uses pipe join topology involving two Web services, left and right. Here the output

of left Web service is used as input to the right Web service. We assume that left Web service is a *search*, *chunked* and *proliferative* Web service. Whereas, the right Web service provides incremental random access to its data, which means that it takes input from the left Web service and then it provides the data objects which match the input in the form of chunks and in the descending order of scores. That is, it performs selection and ranking over its complete data to provides results related to the input given by the right service. In this scenario, we see how parallel data extraction can be helpful to reduce the total amount of time to compute the top-K join results. We analyse the proposed approach by performing various experiments with different settings of the operating parameters.

In the end we present an analysis of the proposed approaches in the light of the research objectives set for this purpose. We assess if the proposed methodologies meet the objectives set for this research.

3.1 Parallel Rank Join for Web Data Sources

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. Here we address 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 and sorted access to the data.
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 *Naïve 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 3.1 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. Therefore, 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.

The objectives are the following: *i*) to minimize overall data access time. *ii*) to avoid the access to the data that does not contribute to the top-K join results.
Therefore we proposes a parallel rank join operator cPRJ that achieves the above mentioned objectives by using a score guided data pulling strategy. This strategy minimizes the access time by extracting data in parallel from all Web services, while at the same time avoiding the access to data that is not useful to compute top-K join results. This

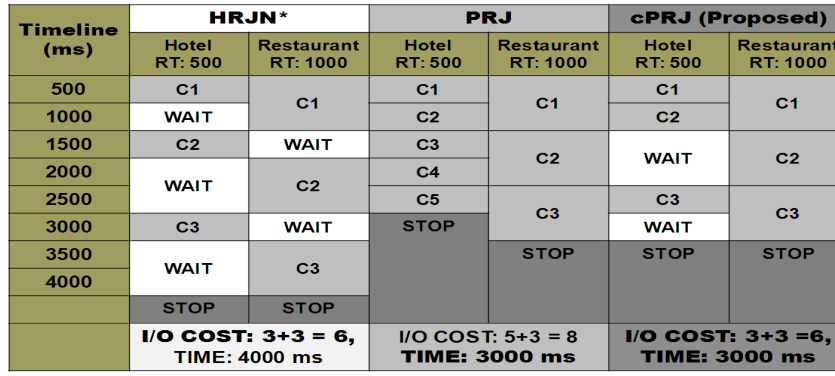


Figure 3.1: Serial Data Access of HRJN* vs Parallel Data Access

is accomplished 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 overall access time, while incurring few extra data accesses, as compared to the state of the art rank join operators.

3.1.1 Methodology

Terminology

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 depth 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.

We propose a new rank join operator cPRJ which involves Web services characterized by non-negligible response time. This operator fetches the data from all the Web services in parallel according to an ad-hoc data pulling strategy and uses a *tight* bounding scheme for the calculation of the local thresholds of the Web services.

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 3.1 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 **??**. We use *tight* bounding scheme to compute the threshold values.

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 3.2. 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 3.1.1. 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 $\sigma(t^{(K)}_{result}) \geq \tau_i$.

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 3.1.1.

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

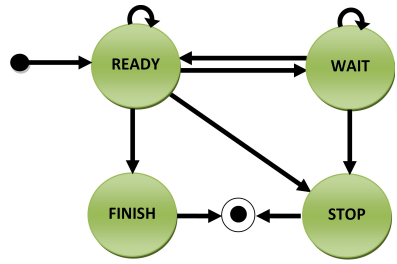


Figure 3.2: The state machine according to which each Web service is manipulated

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 [4] 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. $ttr_j = (chunks \times RT_j) - ET_j$.

State Transitions in the State Machine

The state transitions shown in Figure 3.2 are exemplified below with the help of Figure 3.1.1. 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 3.1.1 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 $\sigma(t^{(K)}_{result})$, 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 3.1.1 shows that after 2100ms the Web service S_3 is put from $Wait$ to *Stop* state as its τ_3 is lower than $\sigma(t^{(K)}_{result})$. 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 3.1.1 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 **??**.

- 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 3.1.1 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 up the other Web services with higher response times. It can be observed in Figure 3.1.1 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.

```
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
1. if(S_result.size ≥ K) then
2.     if(t_result^(k).score ≥ τ_i) then
3.         return 'S'
4. ttr ← 0
5. for j ← 0 to SrcArray.length
6.     if S_j = S_i OR S_j.State ≠ 'R' then continue
7.     diff ← τ_j - τ_i
8.     ttr_j ← 0
9.     if (diff > 0) then
10.        ttr_j ← computeTimeToReach(S_j, diff)
11.    if(ttr_j > ttr ) then ttr ← ttr_j
12. if(ttr ≥ RT_i) then
13.     return 'W'
14. return 'R'
```

Figure 3.3: The setState algorithm

3.1.2 Concurrent Prefetching with cPRJ: A Variant of the Algorithm

It is possible to profile a Web service S_i and identify the number of concurrent calls $S_{i(conc)}$, it supports without any degradation in QoS. Instead of fetching one chunk at a time, the algorithm might issue as many concurrent calls as the Web service permits. This will help in speeding up the data fetching even further, as we shall be able to fetch the data from $S_{i(conc)}$ chunks in RT_i, the time in which we get one chunk with the baseline cPRJ.
In order to achieve this, the *setState* function needs to be changed for the calculation of ttr, by incorporating the number of concurrent chunks extracted by S_i. Also, while issuing the data extraction calls,

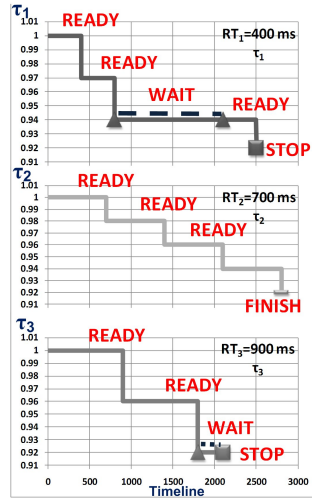


Figure 3.4: Execution of cPRJ with 3 Web services, over timeline against local thresholds.

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. As an example, if $S_{i(conc)}$ is three, and the calculation of ttr shows that τ_i will fall below or equal to $\sigma(t_{result}^{(K)})$ after two fetches, then only two more data extraction calls shall be issued. This variant reduces the time to find the top-K join as compared to the baseline version of cPRJ. However, it incurs in some additional I/O cost because of concurrent data extraction.

3.1.3 Experiments and Results

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 3.3. The real Web services used for the experiments are presented in Table 3.1. 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 3.3. 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 3.1: Real Web services used for experiments

	Web Services	**Type of Information**	**Response Time**	**Chunk Size**
1	`venere.com`	Hotels	900 ms	15
2	`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

Approaches: We compare three algorithms, HRJN*, PRJ and the

Table 3.2: 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

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 outperformed 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.

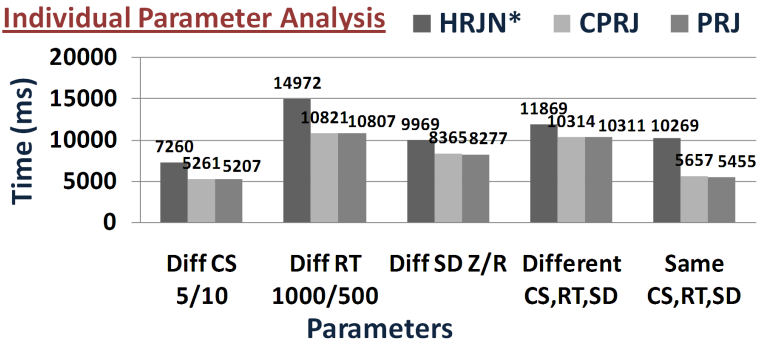
Evaluation Metrics: The major objective of the proposed approach is to reduce the time taken to get the top-K results by minimizing the data acquisition time with the help of parallelism. So, we consider *time taken* as the primary metric for comparing different algorithms. This is the wall clock time, 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* [7], total number of tuples retrieved from all Web services, as other metric for comparing the different algorithms.

Results

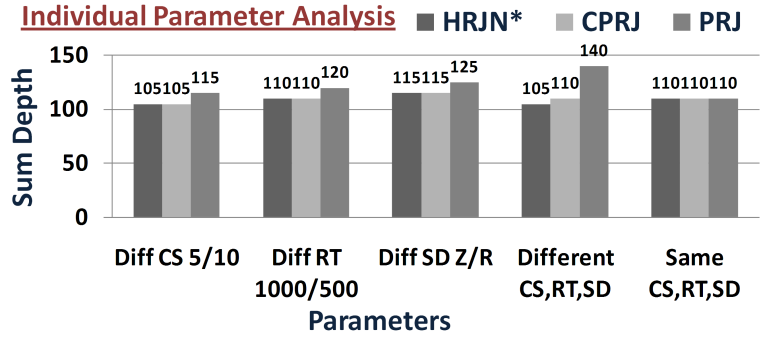
Experiments with Synthetic Data: In Figure 3.5 we show the results of the experiments for CS, RT and SD parameters while joining two Web services. In case of the homogeneous setting of the parameters, i.e. keeping all the parameters to the default values and setting different values for one of the three 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 **??**. The proposed cPRJ algorithm is also based on these three parameters. Figure 3.8(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*. Whereas, Figure 3.8(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 3.1. Figure 3.6(a) shows that both parallel approaches take same amount of time which is 20-25% less than HRJN*. The



(a)



(b)

Figure 3.5: Performance comparison of the algorithms on synthetic data sources for the parameters shown in Table 3.3.

difference in time increase by increasing K. Figure 3.6(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 3.6(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 3.6(d) demonstrate that cPRJ takes 4-11% more I/O cost than HRJN*, whereas, PRJ takes 13-38% more I/O cost than HRJN*.

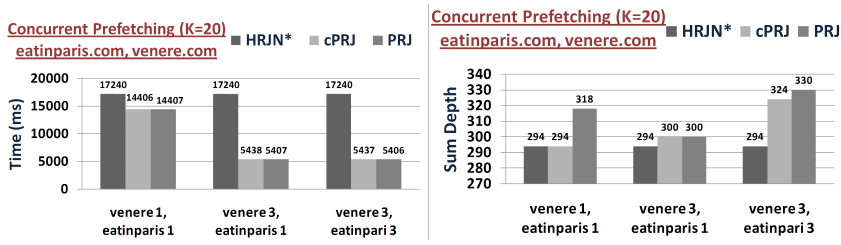
The experimental results also show that other three parameters JS, m and K do not have any impact *alone*. They cannot be responsible for the early termination of a *single* data source. However, if SD, RT and CS have heterogeneous values, and if the overall impact of these values is that they enforce 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 3.6(a) and 3.6(b) show the role of K and Figures 3.6(c) and 3.6(d) show the behaviour of number of data sources m, involved in a query.

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 3.7 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` takes one third or lesser

Figure 3.6: 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`

time to fetch its data from the same purpose. Therefore, when we fetch three concurrent chunks from `verene.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.



(a) (b)

Figure 3.7: 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.

3.2 Pipe Rank Join for Web Data Sources

Consider a pipe join between two services s_L (the $\S$left$\check{\text{T}}$ service) and s_R (the $\S$right$\check{\text{T}}$ service). The input for s_R is (part of) the output of s_L. A motivating example is to query for the best K combinations of events and restaurants in Paris, based on their ratings, by combining an event with the restaurant in the same district. Service s_L might return all the events sorted by score, whereas s_R needs to receive as input a given district and outputs restaurants sorted by score. These are the natural access modes that arise in the context of search computing [5], where answering complex queries might require invoking heterogeneous services, each characterized by its own access pattern.

Let us assume that both services have signature $s_i(A_i, B_i, S_i)$ $i \in \{L, R\}$, where A_i represents the set of attributes specific to service s_i, B_i is the set of attributes used for the join between s_L and s_R, and S_i is an attribute carrying the score for the tuples output by service s_i. The join results are stored in an output buffer s_{BUFF} which has signature $s_{BUFF}(A_{BUFF}, B_{BUFF}, S_{BUFF})$. All the results in s_{BUFF} are sorted in descending order of score value, S_{BUFF}. For the sake of simplicity, w.l.o.g. we consider that both A_i and B_i consist of a single attribute. We assume that s_L enables sorted access (i.e., tuples are output one by one in descending order of the score S_L), whereas s_R enables incremental random access [14] on every distinct join attribute B_R. This means that, for a given value of attribute B_R, the corresponding tuples are output in descending order of the score S_R. Each invocation of a Web service s_i, retrieves a fixed number of tuples, denoted by cs_i and is referred to as *chunk size* for the Web service s_i. In case of s_L, a chunk of data contains the next tuples from it. Whereas, for s_R a chunk retrieves the next tuples for a specific join attribute value B_R. Furthermore, s_i provides a chunk of tuples in a specified time, which is referred to as *average response time* (rt_i). We assume that time to retrieve a chunk of data from s_R i.e. rt_R is same for every distinct join attribute value B_R. The data obtained from s_L and s_R are joined

based on the join attribute values. For simplicity, we assume that score of a join result is assigned by the sum of S_L and S_R, but any other monotonic function would apply.

3.2.1 Methodology

Top-K pipe join provides an output with guaranteed K join results with highest aggregated score. Therefore, similar to the conventional rank-join problem [15], where both services enable sorted access, we need to devise a *bounding scheme* which provides an upper bound on the aggregated score of the unseen join results, and a *pulling strategy* that controls the data extraction from the data sources. In order to describe the join strategy for accessing the tuples available at s_L and s_R we introduce the following notation.

n_L: number of tuples fetched from s_L.

J: number of distinct values for B_L in s_L.

b^j: the jth value for the join attribute B_L ($j = 1 \dots J$).

s_R^j: we consider the invocation of a right source with a distinct join value b^j as a *logically* distinct source.

n_R^j: number of tuples fetched from s_R^j.

$\underline{J}$: is a set that contains all observed distinct join attribute values b^j from s_L.

Bounding Scheme

The bounding scheme computes a threshold value τ which represents an upper bound on the scores of the join results that can be obtained from the unseen data. The threshold value is computed as [14]:

$\tau = \max\{\tau_L, \tau_R\}$, where
$\tau_L = S_L(n_L) + max_{j=1,\dots,J}\{\underline{S}_R^j(1)\}$,
$\tau_R = max\{\tau_R^1, \dots, \tau_R^J\}$,

$\tau_R^j = S_L(n_L^j) + S_R^j(n_R^j)$, for $1 \leq j \leq J$.

Here, $S_L(i)$ represents the score of the i-th tuple extracted from s_L, $S_R^j(i)$ the score of the i-th tuple extracted from s_R, such that $B_R = b^j$, and we consider it a logically distinct data source s_R^j. Here, n_L^j is defined as the position of the first tuple output by s_L for which $B_L = b^j$ or n_L if there is no such tuple among the first n_L. The underlined version $\underline{S}_R^j(1)$ is defined as Şif $n_R^j \geq 1$ then $S_R^j(1)$ else S_R^{max}ŤŤ, where S_R^{max} is the maximum score possible for tuples in s_R. We also define $S_R^j(0) = S_R^{max}$. Apart from this, if the total number of J is unknown then τ_L is always $S_L(n_L)+S_R^{max}$ and τ_R is $max\{\tau_R^1, \ldots, \tau_R^J, \tau_L\}$. Here, $\tau_R^1, \ldots, \tau_R^J$ are the thresholds for all the s_R^j for which b^j exists in $\underline{J}$, that is, b^j has already been observed in s_L. Whereas, τ_L corresponds to the local threshold for s_R^j for which b^j does not exist in $\underline{J}$. τ_L is the local threshold of s_L, and τ_R^j is the local threshold for s_R^j, that is, the threshold of s_R for the join attribute value b^j. A join result at position i in the buffer is guaranteed to be in the set of top-K join results if its aggregated score is greater than or equal to τ i.e. $S_{BUFF}(i) \geq \tau$. The join operator stops when s_{BUFF} has K or more join results having score greater than or equal to τ.

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 [10]. A naïve parallel pulling strategy keeps on extracting data from s_L and from s_R for all s_R^j for which b^j exists in $\underline{J}$, in parallel. It stops data extraction from a Web service s_L or s_R^j, when its respective local threshold becomes equal or below the score of K-th seen join result. This pulling strategy is employed by *parallel Pipe Rank Join* (pPRJ). However, pPRJ may result extracting unwanted data if a Web service s_L or s_R^j stops before the others. E.g. when a Web service stops as its local threshold has reached below the score of, the *then* K-th top join result in the output buffer s_{BUFF} then there is

a possibility that the other Web services having higher local thresholds produce join results with better score values and terminate with an even higher local threshold. Resultantly, the Web service which stops earlier incurs extra data fetches. Though pPRJ approach may compute the join results very quickly yet, it may incur additional I/O cost as compared to our proposed approach *controlled Parallel Pipe Rank Join* (cpPRJ). Therefore, cpPRJ uses a data pulling strategy which not only uses parallel data extraction to reduce the time to fetch the data, but at the same time, controls the access to the unwanted data by pausing and resuming data extraction from the Web services. The services are paused and resumed using a state machine which is the same state machine as used in parallel rank join algorithm cPRJ. 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 3.2. We consider every s_R^j a *logically* distinct Web service. Thus, we have J right Web services. Hence, we transform the problem of pipe join to a parallel rank join problem [2]. However, in a parallel rank join, the data extraction from all the data sources starts simultaneously. Conversely, pipe join begins with data extraction from s_L, and while processing each chunk of data obtained from s_L, if it observes a new join attribute value b^j, then it starts data extraction from s_R for b^j. In other words, it starts data extraction from s_R^j when b^j is observed for the first time in the data obtained from s_L. So, unlike parallel rank join, data extraction doest not start simultaneously from all data sources. These Web services are assigned a particular state after processing the data extracted for a chunk. The *Ready* state means that more data should be extracted from this Web service. The $Wait$ state means that the data extraction from this Web service should be paused. 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. Like in cPRJ algorithm, mainly the state machine behaves in a similar fashion. However, there is a need to customize the execution of this

state machine to make it useful in the case of pipe join. We present these customizations in the following text. On retrieving a chunk of tuples from Web service s_i the following operations are performed in order:

Its local threshold τ_i is updated and global threshold is updated accordingly.

A snapshot of the system consisting of the tuples retrieved in the current chunk, n_L and n_R^j which are the number of tuples extracted from s_L and all s_R^j, is captured and is added into a priority queue for data processing. This priority queue is ordered based on the observed local thresholds of the Web services.

If this is the last chunk then s_i is put to $Finish$ state and τ_i is set to 0.

If $\tau_i \leq S_{BUFF}(K)$, then S_i is put to $Stop$ state.

Otherwise, next data extraction call for s_i is issued if all previously extracted chunks for this Web service have been processed.

However, if the data processing queue still holds a chunk for this Web service then the next data extraction calls is issued if the assessment of the state for this Web service suggests it to be in *Ready* state, failing to which it is put to $Wait$ state. Unlike the cPRJ algorithm cpPRJ algorithm carefully captures the snapshot of the current state of execution. Each snapshot is captured in a synchronous way, which helps avoiding the duplicates in computing the join results. Whereas, the ordering of the captured snapshots helps in increasing the possibility of high scoring join results earlier [3]. Furthermore, the entries in the priority queue are processed asynchronously, and every time the top element of the priority queue is extracted for processing. The data processing time mainly depends upon the number of tuples to be joined together, and number of Web services for which the state needs to be assessed. If the total time required to process the data of all the Web services in *Ready* state is larger than the average response time of a Web service,

then the queue will pile up and it may contain more than one snapshots for a particular Web service. Therefore, in this case, the next data extraction call is issued after the assessment of the state of the Web service as *Ready*. A particular snapshot entry in the queue is processed as follows:

The computation of joins for the data in the chunk based on the information captured in the snapshot for every Web service.

Each Web service s_i, that is in *Ready* or $Wait$ state, is checked and is put into *Stop* state, if $S_{BUFF}(K) \geq \tau_i$.

All Web services in $Wait$ state are checked if they should be kept in the same state or should be put to *Ready* state for further data extraction.

However, all the state transitions are conducted in similar way as of cPRJ algorithm. In fact, as mentioned above, we consider the data extraction for every distinct join attribute value from s_R as a logically distinct Web service. Hence, we set the state for each of these Web services.

Initialization of Data Extraction: The other difference between parallel rank join and pipe join is the initialization of data extraction from the Web services. We strat extrating the data for a particular join attribute values from the right Web servie s_R, when this join attribute value appears for the first time int s_L. Therefore, data extraction from any logical right Web service s_R^j is started and it is put to *Ready* state when it is encountered for the first time in any tuple of s_L. However, before the start of data extraction the respective local threshold τ_R^j is compared with the K-th highest join result in s_{BUFF}. If $S_{BUFF}(K) \geq \tau_R^j$, then the data extraction is not started from s_R^j. This is only possible while processing the last chunk from s_L, before it is put into *Stop* state. It certainly helps in avoiding unnecessary data extraction calls and hence, reduces the overall I/O cost.

3.2.2 Limiting Number of Concurrent Accesses for Right Web Service: A Variant of the Algorithm

In the baseline algorithm we assume that we can issue J concurrent data extraction calls to s_R, that is, we can extract data from s_R for all the distinct join attribute values simultaneously. However, the number of concurrent calls depends upon many factors: *i*) total number of distinct join attribute values; *ii*) number of concurrent calls allowed to a particular client by the Web server hosting the Web service. This can be identified by profiling of the hosting Web server; *iii*) as a physical rank join operator in a database server, the number of concurrent data extraction calls allowed on the same data source or Web service by the database server; *iv*) number of maximum concurrent data extraction calls allowed for a query being processed.

Therefore, based on all these factors we identify number p, which is the maximum allowed concurrent data fetching calls to the right Web service s_R, hence, we extract data from s_R for only p distinct join attribute values concurrently. If $p \geq J$, then the baseline version of the algorithm works as it is. Otherwise, we have to modify the algorithm to ensure that no more than p concurrent data extraction calls should be issued to s_R. In order to restrict the number of concurrent data extraction calls to p, the baseline algorithm is modified in the following manner:

A list l_{ACT} is created which stores all b^j for which data is currently being extracted from respective s_R^j.

When a new join attribute value b^{new} is encountered from s_L and if it is assigned *Ready* state after the initial threshold check, then the algorithm checks if the total number of elements in l_{ACT} are less than p, then it starts data extraction from s_R for b^{new} and adds its corresponding index to l_{ACT}. Otherwise, it is not added to l_{ACT}.

At the completion of data extraction for s_R^j, it is removed from l_{ACT}, and new local threshold τ_R^j is computed and chunk data along with other snapshot information is pushed into the data processing queue

for further processing.

Now, from all the s_R^j which are in *Ready* state and their respective b^j is not in l_{ACT} the algorithm chooses the one with the highest local threshold and starts the data extraction for it from s_R and adds it to l_{ACT}. It continues to start new data fetches until l_{ACT} contains p elements, or there is no s_R^j which is in *Ready* state and is not in l_{ACT}.

Intuitively, the restriction that only p concurrent data extraction calls can be issued to s_R should result into decreasing the I/O cost as compared to the baseline version of the algorithm. The reason is that sPRJ also optimizes the I/O cost by extracting data from only one data source at a time by choosing the most suitable data source every time [14]. However, this restriction may increase the total time if there are more than p candidates for data extraction but they are not allowed to extract data concurrently due to this limit.

3.2.3 Experiments and Results

Experimental Setup Data Sets: We have conducted the experiments based on the motivational example provided in the introduction using both synthetic data, and real Web services. Many diverse synthetic data sets are generated using different values of main operating parameters shown in Table 3.3. We have used uniform and Zipfian score distributions for generating the synthetic data. We have also used both score distributions alternatively to assign scores to the tuples for different b^j to generate tuples for respective s_R^j. We have used Yahoo! Local to conduct the experiments on real Web services. We get the data for events in a city and use it as s_L and then we use each newly observed area code as a distinct join attribute value j and start extracting restaurants located in it, thus forming the respective s_R^j.

Approaches: We compare three algorithms: sPRJ, pPRJ and the proposed cpPRJ. An important consideration is that sPRJ cannot be beaten in terms of I/O cost, whereas pPRJ cannot be out-

performed in terms of time taken, provided the time taken for joining the data is negligible.

Table 3.3: Operating Parameters

Full Name	**Parameter**	**Tested Values**
Number of results	K	50, 200, 500
Number of distinct join attributes in s_L	J	12,16,20,21,34,100
Score Distribution	SD	Uniform Distrib., Zipfian Distrib., Mixed
Response Time	rt (rt_L/rt_R)	500/300, 600/500, 900/500
Chunk Size	cs (cs_L/cs_R)	10/10, 10/15

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* (wall clock time) as the primary metric. The reduction in time is obtained by compromising on possibly some unwanted data extraction. Therefore, we consider *sum depths* [7], total number of tuples retrieved from all Web services, as an indicator of the I/O cost. For fairness, we compute these metrics for 10 different cities in case of real Web services, and over 10 different data sets for synthetic data, and report the average.

Results Experiments with Synthetic Data: In Figure 3.8 we show the results of the experiments for main operating parameters of cs, rt and score distribution (SD). Figure 3.8(a) shows that for all three parameters time taken by both parallel approaches is very close. More precisely, both the proposed cpPRJ and pPRJ take almost 80% lesser time than sPRJ for the experiments conducted with different SD and different cs values. Whereas, both cpPRJ and pPRJ take 56% and 49% lesser time than sPRJ for the experi-

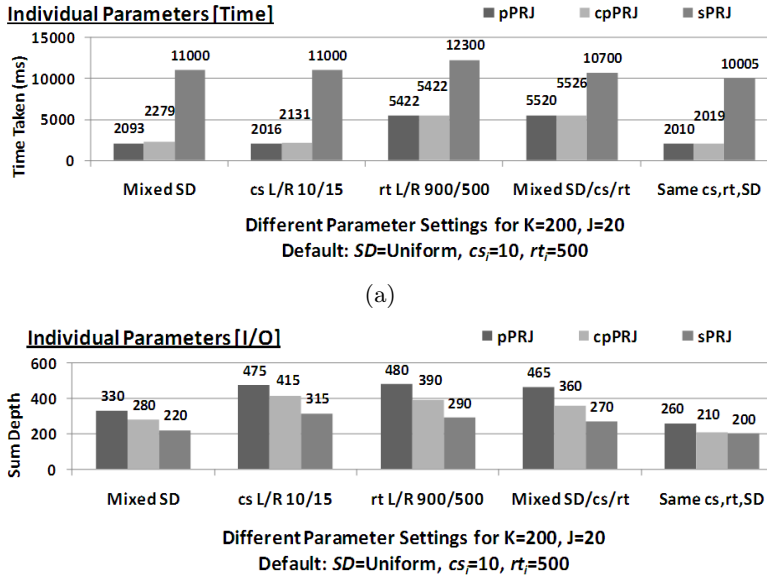


Figure 3.8: Comparison of the algorithms using synthetic data sources for cs, rt and SD parameters.

ments conducted with different values of rt and for the experiment while using different values for all three parameters, respectively. Figure 3.8(b) shows that for the experiments conducted with different values for one of cs, rt and SD pPRJ incurs 50%,50%, and 72% more I/O cost than sPRJ, respectively. Whereas the observations for the corresponding parameters for the I/O costs show that the proposed cpPRJ approach takes 27%,31%, and 34% more I/O cost than sPRJ. However, If we take different values for all these differences in the parameter values in one scenario then pPRJ incurs 72% more I/O cost than sPRJ and cpPRJ costs 33% more I/O cost than sPRJ.

Thus, the proposed cpPRJ approach minimizes the access to the unwanted data access while keeping the total time to compute top-K

joins very close to pPRJ, which is optimal in terms of time. Therefore, like [2] we observe that cs, rt and SD are the parameters whose different values among different Web services result into early termination of a particular Web service. We also observed in [2] that similar values of these parameters result into identical behaviour of both parallel approaches and both of them not only take much lesser time than sPRJ but also take the same I/O cost as of sPRJ. However, Figure 3.8 shows that considering pipe join, the behaviour with the similar values of cs, rt and SD among all Web services is not as in [2]. The reason is that in the context of a parallel rank join, the data extraction from all the data sources is started simultaneously, therefore, the decay in the respective local thresholds, in this particular scenario, is *ideally* similar and all of them reach the *Stop* state simultaneously. However, in the context of a pipe join, the data extraction from s_R^j is subject to the appearance of b^j in s_L, thus, the data extraction from all s_R^j does not start at the same time. Therefore, even with the same cs, rt and SD values for all data sources, their respective local thresholds have different values, unlike in the parallel rank join. The proposed cpPRJ approach deals with the different starting times in a smart way and it brings all the local thresholds to the same value by putting some Web services in $Wait$ state. Thus, its I/O cost is very close to that of the sPRJ. Whereas, the pPRJ approach is unable to bring the local thresholds alike and hence results into extra I/O cost. But, the time taken by both parallel approaches still remains the same.

Real Web Services: We performed experiments with real Web services for different values of K, and different values of J, the number of distinct join attribute values in s_L. Figure 3.9(a) shows that pPRJ takes 65-69% lesser time than sPRJ, whereas, cpPRJ takes 53-62% lesser time than sPRJ for different values of K. Figure 3.9(b) shows that for different values of K the I/O cost incurred by proposed cpPRJ is 0-17% more than ideal sPRJ, whereas, pPRJ extracts 18-48% more data than sPRJ. Figure 3.9(c) shows that for pPRJ takes 62-70% lesser time than sPRJ, whereas, cpPRJ takes 47-55% lesser time than sPRJ for different values of J. In terms of

the I/O cost, Figure 3.9(d) shows that cpPRJ extracts 0-7% more data than sPRJ, whereas, pPRJ extracts 20-37% more data than sPRJ, for different values of J.

Figures 3.9(e) and 3.9(f) show the results of the experiment using Yahoo! Local for execution of the example query for 10 different US cities to get 50 top join results. The results reveal that pPRJ takes 66% lesser time than sPRJ and the proposed cpPRJ takes 51% lesser time than sPRJ. Whereas, pPRJ takes 16% more I/O cost than sPRJ and cpPRJ takes the same amount of I/O cost as of sPRJ.

As a whole, the overall performance of cpPRJ is much better than pPRJ as it has I/O cost close to that of sPRJ, and it takes slightly more time than pPRJ, whereas, pPRJ has higher I/O cost than both cpPRJ and sPRJ and the time taken by both parallel approaches is very close, especially from the end user's perspective.

Restricting the Concurrent Calls on s_R: We have performed experiments using the synthetic data sources while setting different values for the parameter p, which is the maximum allowed concurrent data extraction calls on s_R. The total number of distinct join attributes values J, in the generated s_L are 100. Figure 3.10 shows the comparison of the I/O and time costs for cpPRJ with different values of p, where $p \leq J$. We also include in the comparison the behaviours of sPRJ, and pPRJ with $p=J$. The results reveal the fact that lower values of p result into more similar behaviour of the proposed cpPRJ to that of sPRJ both in terms of time and I/O costs. The other interesting observation is that the difference in I/O cost and time are not significant for $p \geq 20$, which means that during these experiments the maximum number of concurrent data fetches for s_R^j is for nearly 20 distinct b^j values while using cpPRJ. Whereas, for the rest of Web services from which the data extraction has been started have either been *paused*, *stopped* or $finished$ by being put into $Wait$, $Stop$, or $Finish$ state, respectively.

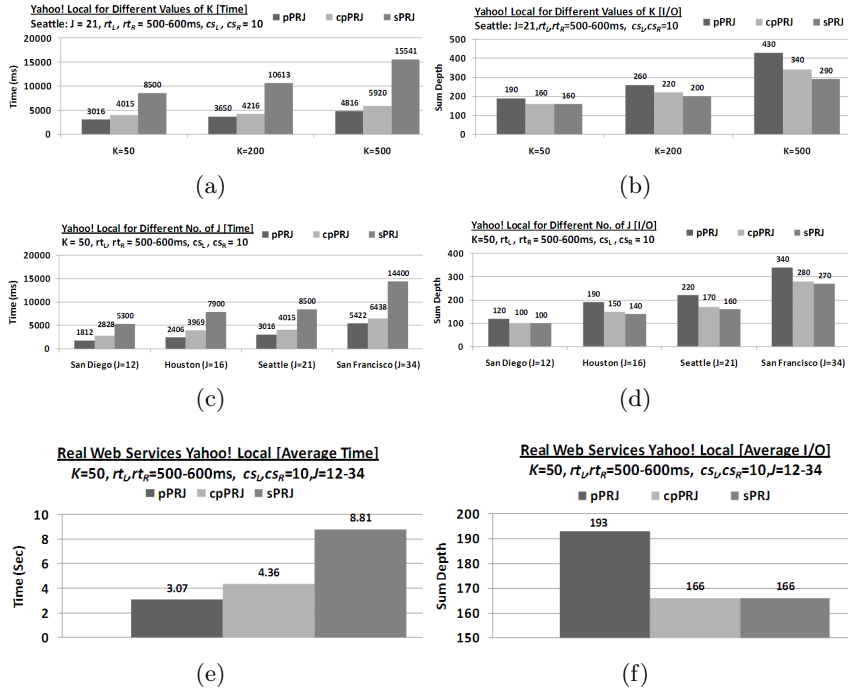
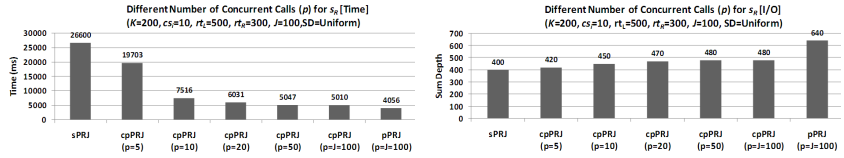


Figure 3.9: Performance of the algorithms with real services, Yahoo! Local. Figures (a) and (b) are for the experiments with different values of K. Figures (c) and (d) show results for experiments with different values of J in s_L. Figures (e) and (f) show the comparison of average time and average I/O costs respectively, based on the results of the experiments for 10 different cities using Yahoo! Local.



(a) (b)

Figure 3.10: Performance comparison of the algorithms on synthetic data sources for different values of p, the number of maximum allowed concurrent data fetches on s_R.

3.3 Discussion

In this chapter, we have presented parallel data access for parallel and pipe rank join topologies. We presented that in the context of Web services as data sources, it is imperative to come up with parallel data access mechanism so as to reduce the time to fetch the data, as the time to process the data is negligible as compared to the time to fetch the data. Therefore, the data fetching is a bottle neck in the context of Web based data sources. We also highlighted existing approaches for parallel and pipe rank join work in a serial manner. They fetch the data, process it and then choose the next data source from where the data should be fetched. This serial data access helps in getting the top-K join results in optimal time. However, at the same time they overlook the bottle neck in the context of Web services, which is the time to retrieve the data. The main assumption of the existing serial topologies is that time to fetch the data is negligible, which is totally opposite in the context of Web services. We also highlight that care-free usage of parallel data extraction may result into a high I/O cost, in almost every case in pipe join topology and in most of the cases in the parallel join topology, especially, when there are more than two Web services involved in a parallel join. Therefore, we have presented parallel and pipe join topologies for processing the rank joins while using parallel data access, which keep compute the join results very quickly and avoid accessing the data that does not contribute to computing top-K join results. We have also presented a thorough experimental analysis of both the approaches with main operating parameters. The results of the experiments show that our proposed parallel data extraction approaches for parallel join topologies help in computing the top-K join results in quick time and in some cases result into slight overhead in the I/O cost as compared to the I/O optimal approaches. Whereas, in case of pipe rank join topology with parallel data access, we compute the join results in quick time, however, we incur slightly extra I/O cost in many cases. Thus, it satisfies our objectives for computing the top-K join results using

parallel and pipe topologies efficiently both in terms of time and in terms of I/O cost.

4 Provisional Reporting for Rank Joins using Probability

One of the contributions of this thesis is to increase the efficiency of reporting and computing top-K join results while using probability. Here, we present a methodology to report the top-K join results with probabilistic guarantees, during the execution of a top-K algorithm. In particular, we use rank join algorithm [**?**] and we are interested in studying its progress to compute top-K join results. We adopt probabilistic measures so as to compute and report the obtained join results with certain probibilistic guarantees instead of reporting them with deterministic guarantees. Such a functionality can be beneficial in terms of reducing the runtime and I/O costs of top-K computations, and also helps in reporting the join results in a quick time to the user.

It is important to note that the proposed approach addresses the top-K queries which involve certain data and use approximation methods to report the join results. The motivation of such top-K join algorithms is to reduce the computational cost, I/O cost and time to compute the top-K join results. In the next section we define the problem settings. In section 4.2 we present the existing approaches which are similar to our approach and discuss the need of our approach. We present our proposed methodology in Section 4.3. We analyse our approach with the help of experimental study in Section 4.4. Lastly, we conclude this chapter with a discussion, presented in Section 4.5.

4.1 Problem Definition

Several algorithms have been introduced in literature to process top-K queries efficiently [?, ?, ?, ?, ?, ?, ?, ?, ?]. The majority of such algorithms involve computations which are exhaustive, that is, the algorithms stop when there is absolute certainty that the correct top-K results have been identified. Among these algorithms is hash rank join algorithm [] presented by Ilyas et. al. This algorithm computes top-K joins from data sources containing different set of objects, which are joined based on a join predicate, e.g. equivalence of the join attribute value. The objects in both lists are in descending order of the scores. The data access to these objects is sequential, i.e. an object at depth d will be accessed before the object at depth $d + 1$. During the execution this algorithm computes join results and stores the top-K join results in an output buffer. It is important to note that this algorithm does not compute partial join results. Like other rank join algorithms, this algorithm also maintains a bound or threshold value, and reports the join results which have aggregated score greater than the bound.

Figure 4.1 shows the a snapshot during the execution of HRJN* algorithm, on two data sources s_1 and s_2, for K = 5. Let us assume that the data in the dark text has been extracted from both data sources. The join results formed with this data are stored in the output buffer s_{buff}. The score aggregation function is this case is linear addition, and we can see the aggregate score of the join results in s_{buff}. Furthermore, based on the score aggregation function the threshold or bound is 1.97. Hence, at this stage, the first two join results in s_{buff} can be reported to the user as the top join results with absolute certainty. However, at the same time we can observe that in order to be sure that the join results in the output buffer are the real top-K joins or not, we have to bring the threshold less than or equal to 1.94, the score of K-th join result in the output buffer. This is possible if we extract 2 more objects from s_1 and 1 more object from s_2. However, even after extracting these tuples

we shall not be able to find any new join result having score greater than the K-th join result in the buffer. Therefore, we argue that there should be a method to compute the probability with which an unreported join result in the output buffer will remain among the final top-K join results. If this probability is high enough (e.g. greater than 0.95) then we can report this join result to the user even if its score is below the current threshold value.

S_1		
ID	Join Attribute	Score
$S_1(1)$	a	1.0
$S_1(2)$	b	0.99
$S_1(3)$	c	0.98
$S_1(4)$	a	0.97
$S_1(5)$	d	0.95
$S_1(6)$	b	0.93
$S_1(7)$	a	0.92
$S_1(8)$	c	0.90

S_2		
ID	Join Attribute	Score
$S_2(1)$	b	1.0
$S_2(2)$	c	0.98
$S_2(3)$	a	0.97
$S_2(4)$	b	0.96
$S_2(5)$	d	0.94
$S_2(6)$	e	0.92
$S_2(7)$	c	0.91
$S_2(8)$	a	0.89

S_{buff}		
ID	Join Attribute	Score
$S_1(2)$-$S_2(1)$	b	1.99
$S_1(1)$-$S_2(3)$	a	1.97
$S_1(3)$-$S_2(2)$	c	1.96
$S_1(2)$-$S_2(4)$	b	1.95
$S_1(4)$-$S_2(3)$	a	1.94

Figure 4.1: Snapshot of HRJN* execution.

It is important to note that such situations arise in the top-K queries which have long running time, e.g. the queries which involve very large data sets and the value of K is also high. Similarly, while using the Web services as data sources the run time of the queries is long, as there is a non negligible time associated with each data extraction from a Web service, and data extraction time overwhelms the data processing time in such cases. Therefore, the joins results in the buffer have to wait for these time taking next data extractions to be reported. Hence, an approach which computes the probability of the unreported join results to be among the final top-K join results will help reporting every join result in a quick time.

4.2 Related Work

The description of the problem clearly suggests that it falls in the certain data and approximate methods category in the taxonomy of the top-K join joins, as presented in Chapter **??**. There is some existing research work in the same category with various different settings. As an example Theobald et. al., [**?**] presented an approach

for probabilistic top-K query evaluation. Their approach is specifically targeted to the TA-Sorted algorithm, which maintains best and worst possible scores of the objects based on partially computed total score. The main idea is to compute the probability for a newly seen object with which it may appear among the actual top-K results. Now, if this probability is below a given threshold probability, which is provided by the user, the object is discarded and is not considered in future. This helps in reducing the number of possible objects to be processed during top-K query evaluation. Moreover, by carefully maintaining bounds for the scores of the most promising objects that have been encountered the algorithm may probabilistically decide to terminate earlier than the regular TA-Sorted deterministic computation. The empirical evaluation of the algorithm presented in [**?**] shows that this algorithm performs well in practice.

The approach presented in [**?**] has some objective similarity to our work, however there are many differences. Firstly, it focuses only on discarding the candidate objects which are partically seen and their score bounds suggest that they are unlikely to be among the actual top-K results, therefore, it is not directly applicable to the algorithm that only considers complete joins like HRJN* and the TA. Furthermore, this approach discards or neglects the objects by analysing probability value that a discarded/unseen object is not in the top-K tuples, independent of the number of unseen objects in the data sources. Therefore, the results are the same whether there are 100 or 100,000 unseen objects.

Another similar approach which provides anytime behaviour to compute top-K join results is presented in [**?**] and is futher improved in [**?**]. An anytime algorithm is an algorithm whose quality of results improves gradually as computation time increases [**?**]. In [**?**] the authors present a probabilistic approach which seeks to report at any point of operation of the algorithm the confidence that the top-K result has been identified. Such a functionality can be a valuable

asset when one is interested in reducing the runtime cost of top-K computations. This approach covers TA and TA-sorted algorithms. However, it is different from our defined problem for two main reasons: firstly, it mainly computes the probability with which the current *complete* result set is the actual top-K result set, whereas, we want to compute the probability for each *individual* unreported join result with which it may appear in the final top-K results. Secondly, both in TA and TA-sorted algorithms we deal with the lists which contain same data objects, whereas, we focus our problem for HRJN* algorithm which deals with two data sets having different objects which are joined together based on the equivalence of a join attribute value. Lastly, our objective is to report an unreported join result as quickly as possible after its creation based on probabilistic measures, instead of keeping it in the output buffer so as to report it with deterministically.

Recent work [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] on probabilistic ranking of data, is orthogonal to the work presented here. The model assumed in these works is that of incomplete data and the probabilistic framework is based on possible worlds semantics. So, their problem setting involve uncertain data and approximate results. In contrast,we assume certain characteristics in the data, based on which, we are interested in computing probability for each unreported join result in the output buffer so as to report it with probabilistic guarantee with which it may exist among the actual top-K results.

There are other approximate query processing methods which use sampling to build a database summary synopsis, and using it to answer queries approximately. The basic settings involve drawing a sample from the underlying data, answering the incoming queries based on the sample, and scaling the results to approximate the exact answers. There are different methods for sampling e.g. uniform or random sampling techniques are discussed in [**?**], histogram based sampling is presented in [**?**] and wavelets are also used as presented in [**?**]. These algorithms are generally used to create an optimized query plan based on the synopsis, e.g. [**?**].

4.3 Methodology

4.3.1 Preliminaries

Consider a query Q which needs two data sources which are Web services s_1 and s_2. These Web services 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 depth 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, $\sigma(t) = f(\sigma(t_1), \sigma(t_2))$. The join results obtained by joining the data from these Web services are stored in a buffer s_{buff} in descending order of their aggregate score. The size of this output buffer s_{buff} is bound by the value K, i.e. the number of required top join results.

Now, we briefly discuss the execution of HRJN* algorithm on the above mentioned data sources. HRJN* algorithm, like other rank join algorithms maintains a bound or a threshold τ, which is the maximum possible score of a join result that can be formed by the

unseen data objects. This threshold τ helps in deterministically reporting the observed join results to the user as a top-K join results in correct order. The computation of the threshold depends upon the score aggregation function. 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. $\tau = \max\{\tau_1, \tau_2\}$.
So, all the currently observed join results having greater value than or equal to τ can be reported to the user, we call them *reported* joins. Let K denote the number of join results for which $\sigma(t) \geq \tau$, then these can be guaranteed to be the top-K. Alternatively, the algorithms stops when $reported$ = K.

*Observation*1 All the join results in the output buffer are stored in the descending order of their scores. Therefore, at a given point in time, during the execution of the algorithm, a join result at position x, which is not among the *reported* joins, can either retain at position x or will be at a new position z, where z *ge* x.

On the other hand the data pulling strategy of HRJN* is that it proceeds in iterations, where in each iteration it extracts the next data object from the data source which has highest local threshold. The ties are broken based on the basis of number of tuples retrieved from the sources and it retrieves data from the data source from which we have retrieved lesser number of tuples. If still there is a tie, then it can be broken arbitrarily. The thresholds of all data sources decreases monotonically because the tuples from the objects are retrieved in the descending order of the score. Therefore, after every iteration, extracting the data from the data source with highest local threshold value, helps in keeping all the local thresholds closer to eachother, which in turn, helps in avoiding any possible extra data fetches. Resultantly, HRJN* results in optimal I/O cost.

Consider the execution of HRJN* algorithm over the data of two Web services s_1 and s_2, which provide their data objects in desending order of scores and in the form of chunks or pages. We know the total number of data objects in each Web service s_i. We also know the join selectivity of these Web services. Therefore, we can compute the total number of expected join results N, from these Web services [**?**]. Consider a snapshot of HRJN* after d iterations. We refer to the number of computed join results at a given stage during the execution of the algorithm to be n, where $n \leq$N.

4.3.2 Algorithm

We know that the basic HRJN* algorithm process the data iteratively. In each iteration it fetches the data from a particular data source, computes the join results based on this newly fetched data, updates the local and global threshold values, and finally, reports the join results to the user which having score greater of equal to the threshold τ. At a given point in time, HRJN* algorithm contains *current* top-K join results in s_{buff}, the output buffer. Some of these are among the *reported* join results. We do not report the rest of the join results in s_{buff} as they may or may not be among the final top-K join results. We modify the basic HRJN* algorithm to compute the probability for each unreported join result with which it may exist among the final top-K joins. If this probability is higher than a given threshold (e.g. 0.90) then we report it to the user. This helps in reporting the identified join results to the user in quicker time and with certain probabilistic guarantee.

In order to compute the probability for a join result to be among the final top-K joins, we take certain assumptions. Firstly, we consider that the score aggregation function is linear additive function, i.e. we simply add the scores of the objects that form a join. Secondly, we also assume that score vectors are uniformly distributed over the $[0,1]^2$ square, i.e. the join results are uniformly distributed among the join space. Figure 4.3 shows the join space for two Web

services which satisfy the requirements for tha application of HRJN* algorithm. It shows that we have extracted a certain amount of data from both Web services, and we have computed the join results from this data. All the join results lie in the region covered by the small square. The diagonal line drawn in the small square depicts the current threshold τ, provided the score aggregation function is linear addition. Now, all the join results which are below this diagonal are deterministically top-K joins. We can see in Figure 4.3 a join result j which has score lower than τ. Let us assume that j is in the output buffer at position y, we represent it as $t_{buff}^{(y)}$. In order to report j with deterministic guarantees the threshold should become lower or equal to the score of j, i.e. $\tau \leq \sigma(t_{buff}^{(y)})$. We may or may not find some new join results with score greater than $\sigma(t_{buff}^{(y)})$. This way the new position for j in the output buffer will be z, we know from $Observation1$ that z *ge* y. Figure 4.3 shows a diagonal line passing through j which defines the minimum threshold value to report j as a certain top-K join result. Now, we can see two triangles outside the small square, and all the join results which may exist in these two triangular regions will have score greater than $\sigma(t_{buff}^{(y)})$.

Let p denotes the probability that an unseen result will have an aggregate score larger than that of $\sigma(t_{buff}^{(y)})$. For aggregation functions like linear aggregation, p will be the ratio between the area of these two triangles to the area of L shaped region corresponding to the unexplored region. This unexplored region is all the area except the small square in Figure 4.3.

Now, we need to compute the probability of observing, respectively, $0, 1, 2, ..., l, ..., Nn$ objects with score larger than $\sigma(t_{buff}^{(y)})$. We can use binomial probability computation to compute the probability of having a join result at each position.
$p(l) = binomial(l, N - n, p)$
Then
$p(z) = p(l + y)$

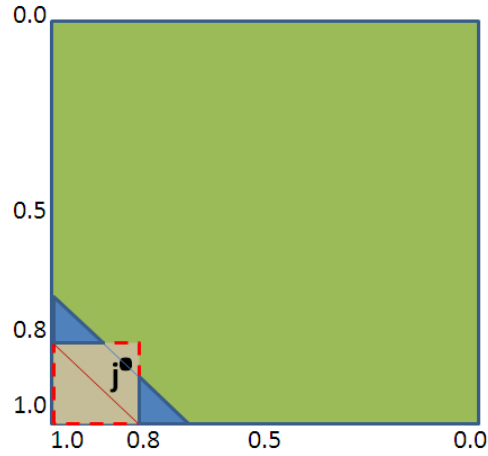


Figure 4.2: Snapshot of Join Space During the Execution of HRJN*

We know from *Observation*1 that $p(z) = 0$, $z < y$.
If we set K be the number of top-K results, the probability that j will be within the top-K join results is $\sum_{z=1}^{K} p(z)$

4.3.3 An Example

Consider the scenario shown in Figure 4.3. Let us assume that $N = 100$, i.e. the total number of join results from the two sources are 100. We have explore the join space shown in the small square and we managed to find 8 join results until this point. Out of them 4 join results had score greater than the threshold and were reported to the user. We want to find 10 top results, i.e. K=10. Now, we want to find the possiblity of the join results which are not among the *reported* joins to be among the top-K. We consider the first un-reported join result from s_{buff} which is $t^{(5)}_{buff}$. We compute the areas of the triangles based on the difference between the τ and $\sigma(t^{(5)}_{buff})$, and thus, compute the probability p. Here p is the ratio between the

total areas of the two triangles and the unexplored area, which is the L shaped region outside the small square. Using this probability value we compute the probabilities of $t_{buff}^{(5)}$ to be at position 5 upto 10. Then we sum all these probabilities to compute the cumulative probability, which is the probability of this join result to be among the final top-K. We report the join result to the user, if this value is greater than 0.90. We can see from Figure 4.3 that in this case the probability that $t_{buff}^{(5)}$ will be among 10 top join results is almost 1.

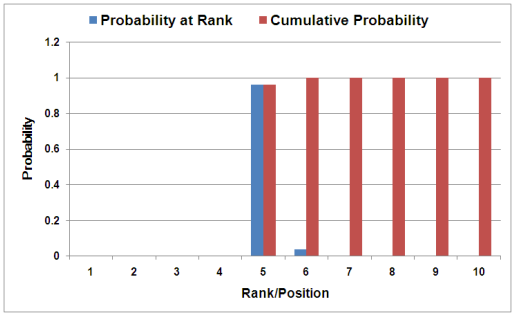


Figure 4.3: Computation of Probability

4.4 Experiments and Results

4.5 Discussion

5 Case Study: Applications in Search Computing

Introduce Search Computing Project.

5.1 Search Computing

Throughout the last decade, Internet search has been primarily performed by routing users towards the specific Web page that best answered their information needs. Major search engines, such as Google, Yahoo and Bing, crawl the Web and index Web pages, highlighting worldwide candidate "best" pages with excellent precision and recall; such ability has proven adequate to fulfill users' needs, to the point that Web search is customarily performed by millions of users, both for work and leisure. However, not all information needs can be satisfied by individual pages on the surface Web. On one hand, the so-called "deep Web" contains information which is perhaps ten times more valuable than what can be crawled on the surface Web; on another side, as the users get confident in the use of search engines, their queries become more and more complex, to the point that their formulation goes beyond what can be expressed with a few keywords, their answers require more than a list of Web pages, and general-purpose search engines perform poorly upon them. According to company's experts, the number of complex queries that are not answered well by major search engines due to their intrinsic complexity is remarkably high and increasing. Many search interactions can be considered as part of a more complex process of expressing goals and achieving tasks, as discussed in the vision paper by Ricardo Baeza Yates. When a query addresses a specific domain (e.g., travels, music, shows, food, movies, health, and genetic diseases), domain-specific search engines do a better job than general-purpose ones; but their expertise is focused upon a given domain. Thus, one can separately find best travel offers and interesting music shows, or conduct genetic analysis and investigate the related medical literature, but can hardly combine information from diverse yet related domains. An expert user can perform several independent searches and then manually combine the findings, but such procedure is cumbersome and error prone. Search computing aims at responding to multi-domain queries, i.e., queries over multiple semantic fields of interest by helping users (or by substi-

tuting to them) in their ability to decompose queries and manually reconstruct results; thus, search computing aims at filling the gap between generalized search systems, which are unable to find information spanning multiple topics, and domain-specific search systems, which cannot go beyond their domain limits. Paradigmatic examples of search computing queries are: "Where can I attend an interesting scientific conference in my field and at the same time relax on a beautiful beach nearby?", "Where is the theatre closest to my hotel, offering a high rank action movie and a near-by pizzeria?", "Who are the strongest candidates in Europe for competing on software ideas?", "Who is the best doctor who can cure insomnia in a nearby public hospital?", "Which are the highest risk factors associated with the most prevalent diseases among the young population?" These examples show that search computing aims at covering a large and increasing spectrum of user's queries, which structurally go beyond the capabilities of general-purpose search engines. These queries cannot be answered without capturing some of their semantics, which at minimum consists in understanding their underlying domains, in routing appropriate query subsets to each domain expert, and in combining answers from each expert to build a complete answer that is meaningful for the user.

Search Computing systems support their users in asking multi-domain queries; for instance, "Where can I attend a DB scientific conference close to a beautiful beach reachable with cheap flights". A system decomposes the query into sub-queries (in this case: "Where can I attend a DB scientific conference?"; "which place is close to a beautiful beach?"; "which place is reachable from my home location with cheap flights?") and maps each sub-query to a domain-expert server (in this cases, calls to servers named "Conference", "Tourism", "Low-Cost-Flights"); it then analyzes the query and translates it into an internal format, which then is optimized, thereby yielding to an optimal plan for query execution; plan execution is supported by an execution engine, which submits service calls to services through a service invocation framework, builds the query results by combining the outputs produces by service calls,

computes the global rankings of query results, and outputs query results in an order that reflects, although with some approximation, their global ranking. These transformation steps are shown in the bottom-left side of Figure 5.1; they are performed by the query mapper, query analyzer, query planner, and execution engine, under the responsibility of a query orchestrator that starts query execution and collects query results. The figure shows that each of the four modules directly accepts user-provided input through suitable interfaces; in this way, prototype implementation in Search Computing can take place bottom-up, by starting with the execution engine, which can execute a given plan, then adding the query planner, which produces the optimal plan for a giver internal query, then adding the query analyzer, which reads an abstract queries, checks that the query is legal, and produces an internal query; and finally adding a query mapper, capable to decompose a multi-domain query into several domain-specific queries. In this book we do not address query mapping, while we address the other steps. Services are made available to Search Computing though a standard format, called service mart; by this term we mean an abstraction that masks the different implementation styles of services and is tailored to the specific need of exposing search services - i.e., services whose primary purpose is to produce ranked lists of results. Moreover, service marts offer a classification of service properties (that represent either the call or the result of a service invocation; given output results may represent the ranking values) and a definition of composition patterns allowing to combine service marts.

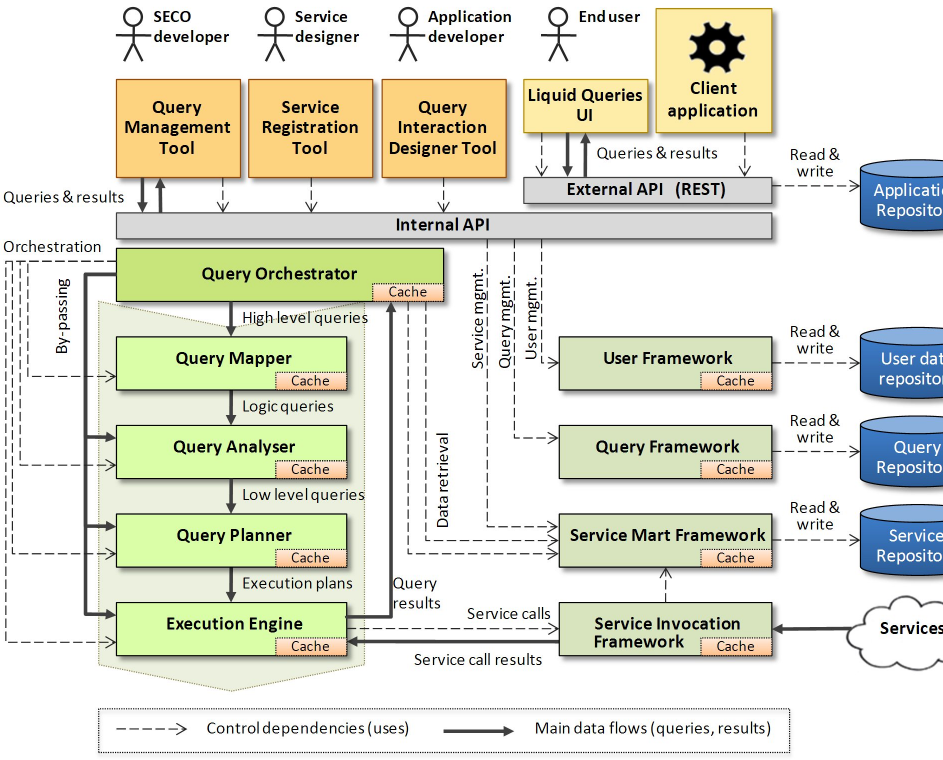


Figure 5.1: Search Computing Architecture.

5.2 Integration with Query Planner and Query Engine

Present how the proposed join topologies can be integrated within the SeCo architecture as join strategies.

5.3 Usage in Liquid Query Processing

This section will explain how the proposed algorithms can be used to process the liquid queries.
Most of the liquid queries can be processed with the help of both pipe and parallel (multi-way) rank join topologies. As an example, a user initiates with finding the apartments for rent from a real estate data source and then she wants to find schools near by. This new data source of schools can be linked to the existing source using pipe join topology, if there is a provision to extract schools by providing the geo-coordinates or the postal code information. Similarly, the same query can be processed using multi-way rank join topology if we can only access the schools using the name of the city.

5.4 Discussion

Summarize and discuss the usage of the proposed algorithms in Search Computing and similar applications.

6 Conclusion

6.1 Discussion

Discuss the main contributions of the thesis while keeping in view the motivations for this work.

6.2 Outlook

The future directions..
dealing with uncertain data.
provisional reporting with more than 2 data sources.
provisional reporting for pipe joins.

Bibliography

[1] N. A., C. Y., S. J. R., L. C, and V. J. S. Supporting incremental join queries on ranked inputs. In *VLDB Conference*.

[2] A. Abid and M. Tagliasacchi. Parallel data access for multiway rank joins. In *ICWE Conference*, pages 44–58, 2011.

[3] P. Agrawal and J. Widom. Confidence-aware join algorithms. In *ICDE*, pages 628–639, 2009.

[4] P. J. Brockwell. *Encyclopedia of Quantitative Finance*. 2010.

[5] S. Ceri. *Search Computing: Challenges and Directions*. Lecture Notes in Computer Science. 2010.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[7] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *SIGMOD Conference*, pages 415–428, 2009.

[8] U. Guntzer, W. Balke, and W. Kiessling. Towards efficient multi-feature queries in heterogeneous environments. *International Conference on Information Technology: Coding and Computing, Proceedings*, pages 622–628, 2001.

[9] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[10] I. Ilyas, W. Aref, and A. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB journal*, 13(3):207–221, 2004.

[11] I. Ilyas, G. Beskales, and M. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM computing surveys*, 40(4):1, 2008.

[12] N. Mamoulis, Y. Theodoridis, and D. Papadias. Spatial joins: Algorithms, cost models and optimization techniques. In *Spatial Databases*, pages 155–184. 2005.

[13] A. Marian, N. Bruno, and L. Gravano. Evaluating top- queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[14] D. Martinenghi and M. Tagliasacchi. Top-k pipe join. In *ICDE Workshops*, pages 16–19, 2010.

[15] K. Schnaitter and N. Polyzotis. Optimal algorithms for evaluating rank joins in database systems. *ACM Trans. Database Syst.*, 35(1), 2010.