Future Generation Computer Systems 113 (2020) 1–12

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Top-*k* query optimization over data services

Abdelhamid Malki [a,*], Sidi-Mohamed Benslimane [a], Mimoun Malki [a], Mahmoud Barhamgi [b], Djamal Benslimane [b]

[a] LabRi Laboratory, Ecole Superieure en Informatique, Sidi Bel Abbes, 22000, Algeria
[b] LIRIS Laboratory, Claude Bernard Lyon1 University, Lyon, France

ARTICLE INFO

Article history:
Received 14 December 2019
Received in revised form 13 May 2020
Accepted 26 June 2020
Available online 4 July 2020

Keywords:
Top-*k* queries
Data services
Data ranking
Query optimization

ABSTRACT

The efficient evaluation of top-*k* queries is crucial for many applications where a huge quantity of data should be ranked and sorted to return the best answers to users in a reasonable time. Examples include, e-commerce platforms (e.g., amazon.com), multimedia sharing platforms, web databases, etc. Most often, these applications need to retrieve data from autonomous data sources. The access to these data sources is carried out through popular Web APIs, such as data web services, to provide a standard way to interact with data. In this context, users' queries often require the composition of multiple data services to be answered. Most of existing solutions for the evaluation of top-*k* queries assume data services to provide both sorted and random accesses to data or only a sorted access. In practice, however, some services may provide only a random access to data, which could impact the performance of the solutions. In this paper, we propose an approach to optimize the evaluation of top-*k* queries over data services. We consider the worst case scenario when services provide only a random access to data. Our approach defines two strategies: *Pipeline Parallel Strategy* and *Necessary Invocation Principle* to reduce the composition processing time and the number of unnecessary service invocations. Conducted experiments showcased the scalability and efficiency of our solution.

1. Introduction

Top-*k* queries, also known as ranking queries, have become so common over the last few years. The capability to efficiently answer Top-*k* queries is now crucial for many emerging applications and information systems that retrieve data from multiple and complex data systems [1]. For examples, web search engines rank and sort websites by their relevance to users' queries and keywords. E-commerce websites and recommendation systems rank their products based on user's profile and preferences in order to increase the purchase rate.

Unlike classical queries that must return the complete set of possible answers that match with the query, top-*k* queries aim to retrieve only the best ranked answers based on a ranking function. When accessing external datasets is required, applications can then be built over data services, a class of Web services that access and query data sources. In this context, user queries are resolved by service compositions, i.e., data services that are relevant to a query are selected and combined together to answer the query [2].

As a motivating example, let us consider a user who wants to plan a trip to USA by searching for less polluted cities; She wants a good quality hotel and a restaurant (in terms of rating), that are situated in the same tourist street; She also wants to walk to the most expensive (ticket cost) amusement park that is highly recommended by its customers (rating). For instance, to search for top-10 combinations (*city, tourist street, hotel, restaurant*) and top-5 combinations (*city, amusement park*) that match the user preferences: city's *pollution level*, hotel *rating*, restaurant *rating*, amusement park *rating* and amusement park *ticket cost*, the above user requirements may be formulated as ranking queries Q_1 and Q_2, over data services described in Fig. 1. These data services can be provided by existing Web data sources.[1] The symbols $\{^b, ^f\}$ denote inputs and outputs of services, respectively.

The top-*k* query Q_1 can be answered by composing the above services as follows: the service s_1 is used to obtain the *Pollution* level for all USA cities without giving any specified input. The free attribute *city* returned by s_1 can serve as input for the service s_2 in order to obtain associated tourist streets *TStreet*. Then, the *TStreet* attribute of s_2 is used to invoke the services s_3 and s_4 and obtain the attributes *Hotel Rating* and *Restaurant Rating* that are used by the user to express her preferences. Finally, only

* Corresponding author.
E-mail addresses: a.malki@esi-sba.dz (A. Malki), s.Benslimane@esi-sba.dz (S.-M. Benslimane), m.malki@esi-sba.dz (M. Malki), mahmoud.barhamgi@univ-lyon1.fr (M. Barhamgi), djamal.benslimane@univ-lyon1.fr (D. Benslimane).

https://doi.org/10.1016/j.future.2020.06.052
0167-739X/

[1] www.aqicn.org for s_1, www.olery.com for s_3, www.opentable.com for s_4, etc.

Data Services	Functionalities
$s_1(City^f, Po^f)$	Returns the list of all cities (*City*) in the USA ordered by their pollution level (*Po*).
$s_2(City^b, TStreet^f)$	Returns tourist streets (*TStreet*) in a given *City* .
$s_3(TStreet^b, Hotel^f, Rating^f)$	Returns *Hotels* name and their *Rating* in a given tourist street *TStreet* .
$s_4(TStreet^b, Restaurant^f, Rating^f)$	Returns *Restaurants* name and their *Rating* in a given tourist street *TStreet* .
$s_5(City^b, AmsPark^f, Rating^f)$	Returns *Amusement Parks* name and their *Rating* in a given *City*.
$s_6(AmsPark^b, TkCost^f)$	Returns tickets cost (*TkCost*) of a given *AmusementPark*.

(a)

$S_1(City^f, Po^f)$

City	Po
cy_1	1.11
cy_2	1.25
cy_3	1.43
cy_4	1.82
cy_5	1.85

$S_2(City^b, TStreet^f)$

City	TStreet
cy_1	T_{11}
cy_2	T_{21}
cy_2	T_{22}
cy_2	T_{23}
cy_2	T_{24}
cy_3	T_{31}
cy_3	T_{32}

$S_3(TStreet^b, Hotel^f, Rating^f)$

TStreet	Hotel	Rating
T_{11}	H_1	10
T_{21}	H_3	4
T_{22}	H_4	7
T_{22}	H_6	6
T_{23}	H_2	5
T_{24}	H_7	8
T_{31}	H_8	10

$S_4(TStreet^b, Restaurant^f, Rating^f)$

TStreet	Restaurant	Rating
T_{11}	R_5	6
T_{21}	R_4	10
T_{22}	R_8	10
T_{23}	R_1	8
T_{24}	R_2	5
T_{31}	R_3	7
T_{31}	R_{10}	9

$S_5(City^b, AmsPark^f, Rating^f)$

City	AmsPark	Rating
cy_1	P_{11}	4
cy_1	P_{12}	5
cy_2	P_{21}	1
cy_2	P_{22}	2
cy_3	P_{31}	2
cy_3	P_{32}	5
cy_4	P_{41}	3

$S_6(AmsPark^b, TkCost^f)$

AmsPark	TkCost
P_{11}	10
P_{12}	6
P_{21}	15
P_{22}	5
P_{31}	20
P_{32}	16
P_{41}	10

(b)

Fig. 1. (a) The ranked data services of the running examples; (b) Sample of the data accessed by the ranked data services.

the 10 combinations with the highest scores (relative to user preferences) will be returned.

Evaluating top-k queries over data services efficiently is a challenging task for two main reasons. First, the best k results cannot be computed until all involved services are invoked and all the join results are generated and sorted. For instance, to evaluate Q_1 and return the best 10 combinations of (*city, tourist street, hotel, restaurant*), we must generate and sort all possible combinations between all cities, tourist streets, hotels and restaurants. However, doing so could be very costly and time consuming since the invocations of some services that appear in the service composition plan could be unnecessary for computing the top-10 combinations. Therefore, avoiding as much as possible of those superfluous invocations is crucial to optimally and efficiently compute the top-k answers. Second, due to data dependencies that can exist between services, a service must wait the complete execution of its predecessor service before starting its execution. For instance, to answer Q_1, the service s_2 cannot be invoked until its predecessor service s_1 has returned all cities. Adopting such a sequential strategy for managing service invocations leads to a long query processing time. Therefore, improving the execution model by applying a parallel strategy is needed to speed up the top-k query processing. In addition, the parallel execution model is motivated by the fact that data services are generally deployed in different external machines/servers.

Contributions. In this paper we present an approach to efficiently process top-k queries over data services by supporting the parallel execution of services and removing superfluous service invocations through the use of heuristics. We first propose a *Pipeline Parallel Strategy* as a parallel execution model for service composition. This strategy may be used to speed up the top-k query processing. Then, we propose the *Necessary Invocation Principle* that minimizes the number of service invocations during the execution of a composition plan. This strategy determines if an invocation is truly necessary to provide the final top-k result. Finally, we propose an heuristic that improves the necessary invocation principle over *parallel join* operators. The proposed heuristic eliminates the single-plan restriction, such that, input tuples are routed differently through the composition plan by generating multiple customized sub-plans.

The remainder of this paper is organized as follows. In Section 2, we give a necessary background about data services and top-k queries. In Section 3, we present our proposed approach for efficiently processing top-k queries. In Section 4, we evaluate our proposed strategies . In Section 5, we compare our approach with the state of art. Finally, in Section 6, we conclude the paper.

2. Preliminaries

In this section, we describe some preliminaries.

2.1. Ranked data services

Unlike relational databases that are originally designed to provide a full access to data sources, data services are exposed with a limited number of access interfaces [3]. A date service s_i can be considered as a virtual table that defines an access pattern to a data source $R_i(a_1, \ldots, a_n)$. This access pattern is modeled using binding patterns notation [4], such as we write $s_i(a_1^{\delta_1}, \ldots, a_n^{\delta_n})$ to denote that s_i is a virtual table of R_i, where $\delta_j \in \{b, f\}$. If $\delta_j = b$, then the value of the attribute a_j must be *bound*/*given* (i.e., $a_j \in$ the set of s_i's inputs I_i), while when $\delta_j = f$, the value of a_j is *retrieved* (i.e., $a_j \in$ the set of s_i's outputs O_i) or it is *free* to be determined or not determined.

In the context of top-k queries, there are two kinds of services: *Ranked* services and *unRanked* services. For the former, tuples returned by a data service s_i are ranked, according to some measure that corresponds to a scoring attribute p_i. The scoring attribute p_i is a *free* attribute of the service $s_i(a_1^{\delta_1}, \ldots, a_n^{\delta_n}, p_i^f)$, with value ≥ 0. However, an *unRanked* service does not have a scoring attribute and returns a set of unranked tuples.

2.2. Top-k queries: Model and composition

In this paper, the class of queries we consider for optimization are top-k *ranking queries* over one or more data services.

Definition 1 (*Top-k Queries Over Data Services*)**.** Let $S = \{s_1, \ldots, s_m\}$ be a set of data services, where, $\exists\ s_i \in S \mid s_i$ has a scoring attribute p_i. Let Q be a top-k query expressed in SQL-Like syntax over S:

Select $\mathcal{X}_p$ *From* $s_1 \bowtie s_2 \bowtie \ldots \bowtie s_n$
Where $\mathcal{B}_1(a_1^{\delta_1}, v_1) \wedge \ldots \wedge \mathcal{B}_l(a_l^{\delta_l}, v_l)$
Order BY $f(p_i, \ldots, p_j)$
Limit k

where $\mathcal{X}_p$ is the set of projected attributes, $\mathcal{B}_1, \ldots, \mathcal{B}_l$ are boolean selection predicates ($==$, $>$, $<$, etc.) applied on no scoring attributes, $f(p_i, \ldots, p_j)$ is a ranking function, k is the number of output tuples with the biggest f values. □

In a top-k query, $f(p_i, \ldots, p_j)$ is a scoring (ranking) function that assigns an aggregate numeric score to each output tuple, and $p_i, \ldots, p_j$, $1 \leq i \leq j \leq n$ are scoring attributes defined over the

involved *ranked* services. In this paper, we are interested only in *monotonic* ranking functions, in which all the scoring attributes positively influence the overall score [5].

The clause *From* describes the join interactions between the involved services (at least one *ranked* service), which allows to define the way with which the services will be composed, i.e., the invocation dependencies between services. For instance, if a *bound* field of s_j is obtained from some *free* attributes of s_i, then there is an invocation dependency $s_i \prec s_j$ (i.e., s_i must occur before s_j in the query plan).

Multiple *composition plans* (i.e., plan of service invocations) can be generated for the same top-k query, where each one respects all invocation dependencies specified in the clause *From*. In addition, a composition plan may include new service invocation dependencies in order to optimize the plan cost.

Definition 2 (*Composition Plan*)**.** Let Q be a top-k query that defines a set of invocation dependencies $\mathcal{PC}_Q = \{s_i \prec s_j, \ldots, s_l \prec s_k\}$ between the involved data services $S = \{s_1, \ldots, s_m\}$. Several Compositions execution plan $C = \{C_Q^1, \ldots, C_Q^n\}$ can be generated for the same top-k query Q, s.t., each composition plan C_Q^i is represented as directed acyclic graph $\langle N, E\rangle$, where:

- $N = S \bigcup \{\bowtie_1, \ldots, \bowtie_n\}$ is the set of graph *nodes* composed of the involved services S and *join* operators
- E is a set of graph *edges* that respects the invocation dependencies in $\mathcal{PC}_Q$, s.t., if there is an invocation dependency $s_i \prec s_j \in \mathcal{PC}_Q$, then s_i and s_j are connected by an arc in the plan, where s_j is the destination and it is invoked after the origin(s_i).

Example. Consider the top-k query Q_1 of the running example which can be expressed in SQL-Like syntax (see Fig. 2(a)). In this query, the free attribute $City^f$ of s_1 is used as an input for the bound attributes $City^b$ of s_2. Moreover, the bound attributes *TStreet* of s_3 and s_4 are provided by the free attribute *TStreet* of s_2. Hence, the set of invocation dependencies between the involved services $\mathcal{PC}_{Q_1} = \{s_1 \prec s_2, s_2 \prec s_3, s_2 \prec s_4\}$. Different possible composition plans can be generated for the top-k query Q_1. For instance, Fig. 2(b) shows a possible composition plan $C_{Q_1}^1$ that respects all invocation dependencies in $\mathcal{PC}_{Q_1}$. The final results is obtained by joining and sorting the outputs of leaf services (s_3 and s_4); and only the best results are reported to the user.

3. Efficient evaluation of top-k queries over data services approach

In this section, we present our approach for efficiently evaluating top-k queries over data services independently of any service composition plan. It is mainly based on two strategies: *pipeline-parallel strategy* and *necessary invocation principle*.

3.1. An overview of the proposed approach

Given a top-k query Q that is formulated over a set of data services (or data APIs) $\{s_1, \ldots, s_m\}$ and characterized by a ranking function $f(p_i, \ldots, p_j)$ and retrieval size k, the objective of our approach is to reduce the processing cost of top-k queries.

Fig. 3 depicts our proposed approach. Users formulate their top-k queries over a service registry in an SQL-Like syntax. The service registry (e.g., Netflix Eureka service registry, Kubernetes/Docker registry) allows to dynamically find the location (i.e., hostname and port) of a data service instance to which an invocation should be sent. In our approach, data services involved in a query composition plan must be executed in a parallel manner (Section 3.2), they then must be available with multiple service instances and deployed on different physical machines. Thus, in order to address this issue, our data services are implemented by using the MicroService Architecture [6] in which services are deployed independently on different physical machines and Cloud VMs, such that each one is *completely autonomous*, and has its own application server (e.g.,Tomcat, JBoss, NodeJs, etc.) and database (e.g., Mysql, MongoDB, Cassandra, etc.). In addition, a data service can expose different instances. These instances may be created and launched dynamically based on different parameters (e.g., load balancing [7,8], etc.). Our approach uses multi-instances concept and allows data services to be represented with multiple instances that can be executed on different physical machines and VMs as shown in Fig. 3.

Once the top-k query Q is formulated, our framework generates the corresponding composition plan C_Q through the *Composition Plan Generation Module*. The composition plan C_Q is obtained by rewriting[2] the top-k query based on the involved data services and the set of invocation dependencies. The invocation dependencies define how the data services will be orchestrated (i.e., *joined* together) and invoked in the composition plan. Thus, we defined two join operators: *parallel join* and *bind join* (the Section 3.2).

Prior to the composition execution, the *Composition Plan Transformation module* is used to perform some transformations of the composition plan in order to improve the necessary invocation principle over parallel join operators (Section 3.3.2 and Algorithm 1). The main idea is to eliminate the single-plan restriction and generate different sub-plans in which the same data service is available with its different service instances.

The *enhanced* composition plan is then executed by the *Composition Plan Execution Engine* by invoking the involved ranked data services based on two different invocations: *Sorted Invocation* and *Random Invocation* (Section 3.3). All invocations are checked by the *Necessary Invocation Principle* component, and only necessary invocations are performed. Unnecessary invocations, whose returned tuples do not have any chance to participate in generating the final top-k result, are avoided.

The invocations of the involved services are scheduled through the *Pipeline-Parallel Execution* component by using the *pipeline-parallel strategy* (Section 3.2). The main element of this component is the *TQEA* algorithm (Section 3.4) that enables to invoke multiple external data services simultaneously in a transparent and parallel way. For each data service s_i of the composition plan, the *TQEA* algorithm launches a process instance that manages the invocation of s_i.[3] Different types of the process instances are scheduled by the *TQEA* algorithm: *Root_Service_Instance*, *Bind_Join_Instance*, *Union_Instance* and *Output_Instance* (Section 3.4). The type of process instance is defined based on the position of the invoked data service in the composition plan and its parent services/nodes (i.e., invocation dependencies).

3.2. Pipeline parallel strategy

While analyzing the state of the art, the most of proposed top-k algorithms [5,14–17] are based on a sequential strategy. However, in the context of using data services, a sequential top-k algorithm needs pointlessly long query processing time, since data fetching exhibit high and variable latency [18,19]. Therefore,

[2] We proposed in previous works [2,9–13] different approaches that efficiently rewrite queries over data services. However, in this paper, we will focus on the top-k query optimization issue.

[3] The aim of the process instances is to launch and manage the pipeline-parallel service invocations by giving input data and then receiving the obtained output tuples. However, the invocations are performed and executed in external servers, in which the service s_i is deployed (i.e., data services are executed in parallel)(see *Multi-Instances Data Service Deployment* of Fig. 3).

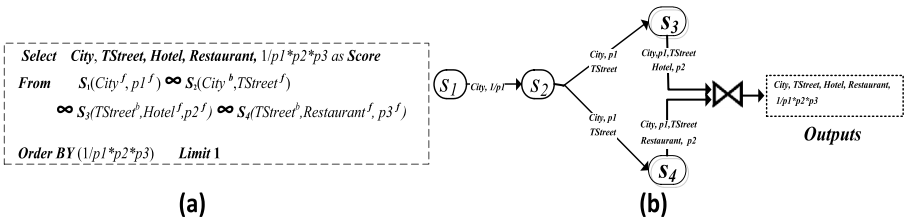


Fig. 2. (a) Top-k query Q_1. (b) Composition plan $C^1_{Q_1}$ of $Q1$.

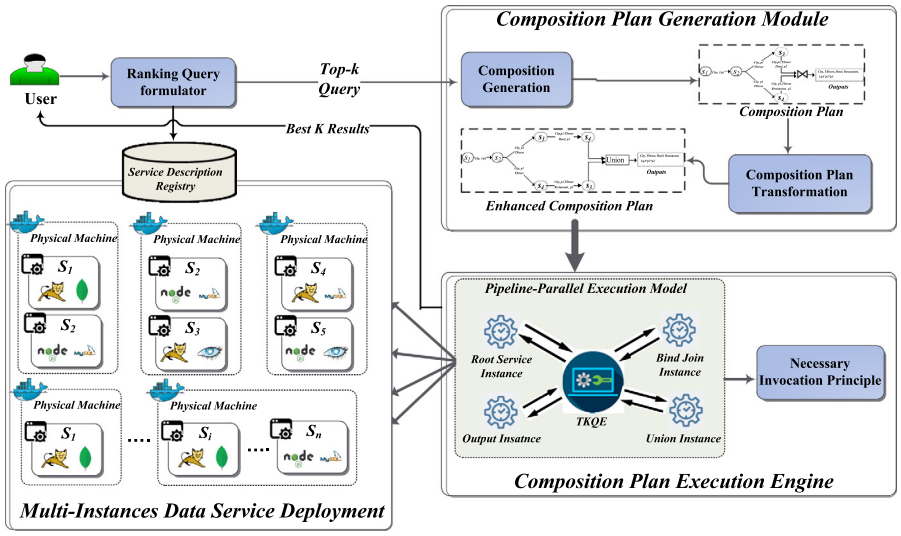


Fig. 3. Top-k optimization framework.

an alternative strategy that invokes data services in a parallel manner must be established in order to minimize the data fetching time by overlapping the waiting times.

In this paper, we use the *pipelined parallelism* query execution model as a first step to speed up the top-k query execution. In *pipelined parallelism*, data already processed by an operator (a service in our case) s_i may be processed by a subsequent operator s_{i+1} in the pipeline, at the same time as the sender operator s_i processes new data [3,20,21]. Based on the set of invocation dependencies, the data services may be invoked (or joined) in two different strategies: *parallel join* and *bind join*.

The *parallel join* is used when there is no invocation dependencies between two services. In this case, both services may be executed simultaneously by dispatching in parallel the same input data (if they have the same parent) to each one, and then performing a join of the outputs of all invoked services. The join result schema contains only the attributes (from all the joined services) that are needed for continuing and completing the service composition execution (e.g., joining the intermediate tuples, select clause, ranking function, etc.). For instance, in the composition plan C_1 (Fig. 2(b)), the data services s_3 and s_4 are invoked in parallel by using *parallel join* operator, since there is no invocation dependencies between them.

Definition 3 (*Parallel Join*)**.** Let $S_{join} = \{s_1, \ldots, s_m\}$ be a set of data services from a service composition plan C_Q that respects the set of invocation dependencies $\mathcal{PC}_Q$ of a top-k Query Q, such that, $\forall (s_i, s_j) \in S_{join}$: $\nexists\, s_i \prec s_j \in \mathcal{PC}_Q$. The services in S_{join} will be executed in parallel, and then joining their outputs. Let $t^{out}(a_1, \ldots, a_p)$ be a joined output tuple, such that, each attribute $a_i \in t^{out}$ is projected out because: (i) a_i was(will be) an input for a data service in C_Q; or (ii) a_i appears in the select clause; or (iii) a_i is used in the ranking function.

The *bind join* is used when there is an invocation dependency between two services. In this case, the services will be invoked in sequence, such that, the outputs of the parent service are used as inputs for the next service. However, the second service may start its execution when some inputs are already available (from the first service), without needing to wait for the complete execution of its parent service, i.e., in pipeline. In cases where a service has more than one direct predecessor, the next service in the *bind join* operator will be preceded by a *parallel join* operator that invokes and joins its predecessor services in parallel way. Thus the obtained joined tuples are used to invoke the second service in a pipeline manner as soon as they are available. In the final results of the *bind join* operator, we combine each output tuple (from the second service) with its input tuples (from the predecessor node), and we project out only the needed attributes, like in the *parallel join* operator. For instance, in the composition plan C_1 (Fig. 2(b)), services s_2 and s_3 will be invoked by using *bind join* operator since there is an invocation dependency between them. Service s_3 is invoked in a pipeline manner with each tuple (*City, Po, TStreet*) returned by s_2. As final output we have the schema (*City, Po, TStreet, Hotel, Rating*) in which all the attributes are needed for the remainder of the composition, e.g., *City* is used for the final join; *Po* and *Rating* are used in the ranking function, etc.

Definition 4 (*Bind Join*)**.** Let s_i be a data service in a service composition plan C_Q that respects the set of invocation dependencies $\mathcal{PC}_Q$ of a top-k Query Q, such that, $\exists\, s_j \prec s_i \in \mathcal{PC}_Q$. Let t^{in} be an input tuple for s_i that is returned by its (their) predecessor(s), such that, each s_i's output is joined with t^{in} in the *bind join* operator and only the needed attributes are projected out.

Executing a service composition plan in a pipelined parallelism manner may surely reduce the overall latency for processing a

top-k query. However, since only the best k objects are required, many service invocations (for both *parallel* and *bind join*) are not needed to compute the final results which increases the I/O cost and thus the run-time query processing. As a solution to this problem, we propose in the next section an approach that enhances the pipelined execution strategy by eliminating undesirable tuples sooner and thus reducing the number of unnecessary invocations.

3.3. Necessary invocation principle

The necessary invocation principle aims to minimize data service invocations during composition plan execution. A data service is invoked if and only if there is a guarantee that its intermediate outputs are useful in generating the final top-k result. A such principle is important in many cases, for instance, when the invocation of a service involves a huge amount of input data tuples, or when the data service is associated with non-negligible response time. In both cases, the time to fetch the data may delay the generation of top-k results.

Given a service composition plan C_Q for a top-k query Q, how to execute C_Q while minimizing the service invocation cost?. Since only the k best objects are requested, complete executing (i.e., invocation of each involved service with every input tuple) may be unnecessarily expensive, since several intermediates tuples are not needed to produce the final top-k results. We distinguish two different invocations according to the binding patterns (access method) supported by the ranked service s_i: *Sorted Invocation* and *Random Invocation*.

Sorted Invocation is available over root ranked services, i.e., all service's attributes are *free* and no input tuple t^{in} is required for the invocation, $s_i(a_1^f, \ldots, a_n^f, p_i^f)$. In other words, s_i is a root service, if $\nexists$ an invocation dependency $s_j \prec s_i \in \mathcal{PC}_Q$ (does not have a parent service). In this case, tuples are accessed sequentially per (tuple/chunk) ordered by the scoring attribute p_i. Sorted invocation can return results in chunks of a fixed size, as the number of root service's results may be generally large.

Definition 5 (*Sorted Invocation*)**.** Let $s_i(a_1^f, \ldots, a_n^f, p_i^f)$ be a root ranked service. A sorted invocation of s_i $invoke(s_i)$ is an invocation that returns a set of ordered output tuples (per chunk), where each output $t^{out}(a_1, \ldots, a_n, p_i)$ is composed of all attributes of s_i. Let t_d^{out} and t_{d+1}^{out} be outputs tuples of s_i at depths d and $d+1$, respectively, then $t_d^{out}.p_i \geq t_{d+1}^{out}.p_i$

However, the *Random Invocation* is performed, when the values of certain attributes must be specified as input to obtain results. In this case, the data service is preceded by other services which give it its input values. A service s_i with the random invocation is formally represented as: $s_i(a_1^{\delta_1}, \ldots, a_n^{\delta_n}, p_i^f)$, where $\exists\ l \in \{1, \ldots, n\} \mid \delta_l = b$, i.e., $\exists\ s_j \prec s_i \in \mathcal{PC}_Q$. In other words, a *Random Invocation* is a *bind join* over ranked data services.

Definition 6 (*Random Invocation*)**.** Let s_i be a ranked service, and let $parent_{s_i} = \{s_1, \ldots, s_{i-1}\}$ be the set of its direct/indirect predecessor services that are invoked before s_i in the service composition plan C_Q. Let $t^{in}(a_1, \ldots, a_m, p_1, \ldots, p_{i-1})$ be an input tuple for s_i with the effective score of each ranked service in $parent_{s_i}$. The invocation of s_i with t^{in}, $invoke(s_i, t^{in})$, is performed as a *bind join* operator that gives the effective score of s_i in a random way, i.e., the output $t^{out}(a_1, \ldots, a_n, p_1, \ldots, p_{i-1}, p_i)$

The cost of a top-k query is largely influenced by the service invocation methods. It is clear that the random invocation is more expensive than the sorted invocation [14,15]. This is explained by the fact that in the sorted invocation, outputs are accessed sequentially ordered by the scoring attribute, while for random invocation, outputs are directly accessed by their bound (input/join) attributes [22]. Controlling and minimizing the number of random invocation is then needed, since they usually have higher costs than sorted invocation, i.e., we should determine if a particular random invocation of a ranked service s_i on an input tuple t^{in}, $invoke(s_i, t^{in})$, is truly necessary to produce the final answer for the top-k query Q.

Consider an intermediate input tuple $t^{in}(a_1, \ldots, a_m, p_1, \ldots, p_{i-1})$ for which we have already invoked the precedence services of s_i, $parent_{s_i} = \{s_1, \ldots, s_{i-1}\}$. Generally, before that t^{in} is fully invoked with s_i and thus with its successor services $next_{s_i} = \{s_{i+1}, \ldots, s_n\}$, the scores $p_1, \ldots, p_{i-1}$ obtained from the evaluated services in $parent_{s_i}$ may give an approximate/maximal query score which defines if the random invocation $invoke(s_i, t^{in})$ is necessary.

For each *Ranked* services s_i, we define a score upper bound $\overline{p}_i$ that represents the maximum possible score of p_i that an output tuple t^{out} can get. We assume that the value of $\overline{p}_i$ can be tracked and estimated by a profiling and statistics component. Let $f(t)$ (or $f(p_1, \ldots, p_n)$) be a ranking function that determines the query score of the tuple t if it is a final output tuple for which all ranked services were invoked (i.e., fully invoked). Thus, we define $\overline{f}_{s_i}(t^{in})$, the maximal-possible query score that t^{in} may eventually achieve, given the attributes scores in $parent_{s_i}$ and scores upper bound in $\{s_i\} \cup next_{s_i}$. Formally $\overline{f}_{s_i}(t^{in})$ is defined as follows:

$$\overline{f}_{s_i}(t^{in}) = f(p_1, \ldots, p_{i-1}, \overline{p}_i, \overline{p}_{i+1}, \ldots, \overline{p}_n)$$

Consequently, our proposed *Necessary Invocation Principle* can be presented by the following property.

Property 1. *Let Q be a top-k query with a scoring function f and a service composition plan C_Q. Let currentTop-k be a set of current top-k results. Given s_i be the next unevaluated ranked service in C_Q with the input tuple t^{in}, then the random invocation $invoke(s_i, t^{in})$ is not necessary to be performed if $\forall\ r \in$ currentTop-k: $\overline{f}_{s_i}(t^{in}) < f(r)$.*

Property 1 helps minimize the of number unnecessary invocations. It provides an "operational" definition to actually determine if a given random invocation is necessary to compute the final top-k objects.

3.3.1. Necessary invocation over bind join

In this part, we show how to use the above Property over *bind join* operator in order to reduce the unneeded random invocations. In the *bind join* operator, a ranked service s_i is invoked with an input tuple t^{in} that is returned by a predecessor node. This predecessor node may be a data service (both root and intermediate service) if s_i has only one predecessor, or it may be a *parallel join* operator if s_i has multiple direct predecessors. In both cases, the necessary invocation principle is directly applied on the *bind join* operator, no matter how the t^{in} was obtained, i.e., before invoking s_i with t^{in}, we check if t^{in} has a chance to be one of the final top-k objects by computing its maximal-possible query score $\overline{f}_{s_i}(t^{in})$ over s_i.

However, when t^{in} is provided by a root ranked service (*Sorted Invocation* style) the maximal-possible query score $\overline{f}_{s_i}(t^{in})$ can be used not only to avoid the unneeded random invocations, but also to stop the execution of the composition and reporting the final best k answers. The composition execution will eventually stop when the current top-k set is fully surfaced by checking the *Stop Condition* over root ranked service. Since, the returned tuples in a *Sorted Invocation* are ordered by their scoring attribute ($t_d^{in}.p_i \geq t_{d+1}^{in}.p_i$), we can then halt its invocation if the maximal-possible query score of a returned tuple at depth d is less than the minimum query score in the current top-k set.

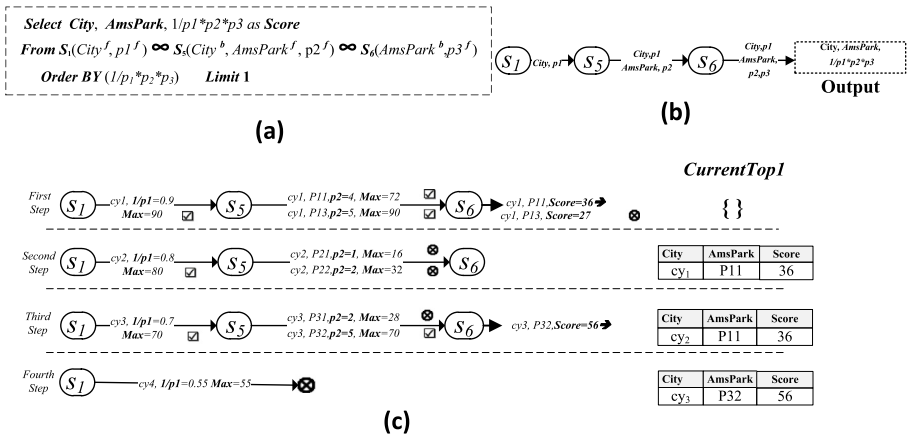


Fig. 4. (a) Top-k query Q_2. (b) Composition plan of Q_2. (c) Execution steps.

Formally, let *currentTop-k* denote the current top-k set for which $\forall\ r \in$ *currentTop-k*: $f(r) > \overline{f}_{s_i}(t_d^{in})$, then these can be guaranteed to be the final top-k results.

Example. Consider the top-k query Q_2 of Fig. 4(a). This query returns the top-1 couple (*City*, *AmusementPark*) based on the ranking function $f = \frac{1}{p_1} \times p_2 \times p_3$ (or $\frac{1}{Po} \times Rating \times TkCost$), i.e., the best Amusement Park in term of *Ticket Cost* and recommended *Rating* that is situated in less polluted city. We assume that the scores upper bound of the used scoring attributes are defined as follows (see Fig. 1(b)): $\overline{\frac{1}{Po}} = 0.9$, $\overline{Rating} = 5$ and $\overline{TkCost} = 20$. The Fig. 4(b) shows the corresponding composition plan in which there are two *bind join* operators: $s_1 \bowtie s_5$ and $s_5 \bowtie s_6$ (i.e., s_5 and s_6 will be invoked with *Random invocation* style). The Fig. 4(c) illustrates the steps for retrieving the top-1 object based on the encapsulated data (Fig. 1(b)). In the first step, the less polluted city [$cy1$] is retrieved from s_1 with a *Sorted invocation* (i.e., s_1 is a root ranked service), and all the next *Random invocations* in the pipeline are performed because the *currentTop*-1 set is still empty. Hence, the *currentTop*-1 set is updated with [$cy1$, $P11$, 36]. In the second step, a new city [$cy2$] is retrieved from s_1. The random invocation *invoke*(s_5, $cy2$) is *necessary* because the maximal-possible query score $\overline{f}_{s_5}(cy2) = \frac{1}{1.25} \times \overline{Rating} \times \overline{TkCost} = \frac{1}{1.25} \times 5 \times 20 = 80$ is higher than the maximal score in the *currentTop*-1 set. The invocation of the next service s_6 with the two tuples retuned by s_5 is not necessary because their maximal-possible query scores (16, 32) are less than the score in *currentTop*-1. In the third step, the same process is applied and the *currentTop*-1 set is updated with a new top-1 object [$cy3$, $P32$, 56]. In the fourth step, the invocation of the service s_5 with the tuple [$cy4$] is not necessary, consequently, the execution of the composition is stopped since the *stop condition* holds. Finally, the object [$cy3$, $P32$, 56] can be safely reported as top-1 answer.

3.3.2. Necessary invocation over parallel join

In the previous section, we discussed how to avoid the unnecessary random invocations over *bind join* operator. This section completes the picture by studying the necessary invocation problem over *parallel join* operators and proposing optimal input tuple scheduling that identify the best way to avoid the unneeded random invocations for parallel services. In this paper, we are particularly interested with the *parallel join* operators that take into account only the random services.[4] Also, we assume that these parallel services have the same predecessor node, i.e., the same inputs data are dispatched to invoke them. To the best of our knowledge, top-k query optimization over relational databases for rank join operators endowed with only random access has not been addressed.

Consider our top-k query Q_1 and its composition depicted in Fig. 2. Q_1 returns the top-1 combination (*City*, *TouristStreet*, *Hotel*, *Restaurant*) based on the ranking function $f = \frac{1}{p_1} \times p_2 \times p_3$, i.e., the highly recommended Hotel and Restaurant that are situated in the same Tourist-Street and in a less polluted City. We assume that the scores upper bound of the used scoring attributes are defined as follows (see Fig. 1(b)): $\overline{\frac{1}{Po}} = 0.9$, $\overline{HotelRating} = 10$ and $\overline{RestaurantRating} = 10$.

Fig. 5 illustrates the first two steps for retrieving the top-1 result. The elimination of unnecessary invocations over *parallel join* operator is almost similar to *bind join* operator. For example, in the second step, the service s_2 returns four tuples {in_1, in_2, in_3, in_4} that are used to invoke s_3 and s_4 (i.e., four invocations for each one). For each input tuple, the maximal-possible query score over s_3 or s_4 is the same, because its exact score over these services is still unknown, e.g., $\overline{f}_{s_3}([cy2, T_{21}]) = \overline{f}_{s_4}([cy2, T_{21}) = \frac{1}{1.25} \times \overline{p}_2 \times \overline{p}_3 = 0.8 \times 10 \times 10 = 80$.

However, all random invocations over s_3 and s_4 will be permitted, while it is clear that most of the join results have a lower final score which does not allow them to be in the top-k set. This is explained by the fact that the exact score of more than one service (i.e., the case of *parallel join* operator) is unknown when computing the maximal-possible query score, which gives a wrong decision about the *Random Invocation* necessity.

In order to improve the necessary invocation principle over *parallel join* operator, we propose an approach that eliminates the single-plan restriction, such that, the input tuples are routed differently through the composition plan by generating multiple customized sub-plans. The main idea is to transform the *parallel join* operator to different *bind join* operators, such that, if there is a *parallel join* between two random services s_i and s_j, then we generate two sub-plans that contain, respectively, the *bind join* operators: $s_i \bowtie s_j$ and $s_j \bowtie s_i$. These sub-plans are executed in parallel, where the common services that are involved in both sub-plans are represented by different and independent process instances that take different input sets. For instance, at the execution stage, the service s_i will be represented by two process instances, one for each sub-plan. However, in *parallel join* operator, each involved service is associated with a single instance that process all input tuples. In addition, the use of *bind join* operator enhances the maximal-possible query score estimation by reducing the number of unknown scoring attribute value.

[4] Invoked with random style, i.e., services that have a predecessor node.

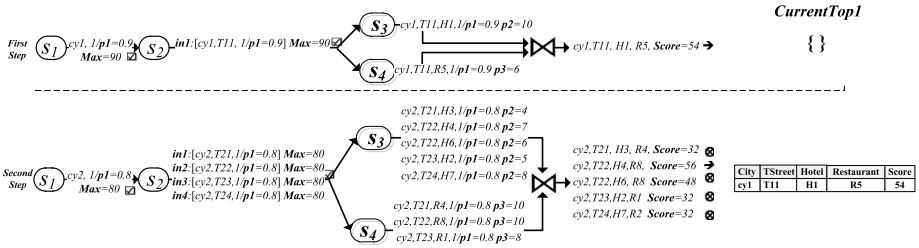


Fig. 5. Execution steps with parallel join operator.

Once the sub-plans are generated, the next step is to define how to route different input tuples to different sub-plans. Let $Input_{Join} = \{t_1^{in}, \ldots, t_n^{in}\}$ be a set of input tuples that are returned by the predecessor of s_i and s_j. Since, there are two sub-plans ($s_i \rightarrow s_j$ and $s_j \rightarrow s_i$) we need to partition the set $Input_{Join}$ into two subsets of tuples, a subset for each route. The input tuples can be partitioned based on different data characteristics and statistical properties: selectivity, score distribution, correlation, etc. In our work, we use a naive strategy that randomly partitions the set of input tuples into m nearly equal subsets, where m is the number of sub-plans.

Fig. 6 shows the enhanced composition plan of the second step of the previous example (Fig. 5). In this plan, the *parallel join* between the services s_3 and s_4 is transformed into two independent *bind join* operators $s_3 \bowtie s_4$ and $s_4 \bowtie s_3$. The set of input tuples is partitioned into two subsets: $\{in_1, in_2\}$ for the sub-plan $s_3 \rightarrow s_4$ and $\{in_3, in_4\}$ for the sub-plan $s_4 \rightarrow s_3$. For instance, in the first sub-plan $s_3 \rightarrow s_4$, the service s_3 is invoked only with two tuples rather than four tuples. Thus, before the invocation of the service s_4 with these two tuples, an invocation was eliminated because the obtained maximal-possible query score are more significant. Finally, we see that the same result was obtained as in the previous composition, but with a reduced number of intermediate tuples and random invocations.

3.4. Algorithms

Algorithm 1 describes how we can eliminate the single-plan restriction, such that each *parallel join* operator is transformed into different *bind join* operators. We start by finding the *parallel join* nodes from the composition plan C_Q (line 1–2). Then, for each *parallel join* node N_i, we firstly get the involved parallel services (line 3) with their parent node N_j(line 4) and we generate the set of *Sub_plans* (line 5), such that, if we have n parallel services, then we must generate n different sub-plans. In the second step, we applying the transformation on the composition plan, such that, all the parallel services are removed (line 6) and the *parallel join* node N_i is replaced by an union node N_i^{union}. Finally, for each sub-plan sp, we add a directed edge from N_j (the parent of the parallel services) to the root service in the linear sub-plan sp and we add a directed edge from the leaf service of sp to the union node N_i^{union}.

The *TQEA* Algorithm (Algorithm 2) describes how to execute a composition plan for a given top-k query, while applying the necessary invocation principle. In the first step, the composition plan C_Q is optimized by using the *Composition Plan Transformation* Algorithm (line 2). In the second step, the execution of the composition plan is started by establishing a *process instance* for each node in C_Q.

The process instances access to the data services/nodes by performing asynchronous invocations[5] in order to allow the top-k query processing algorithm to continue without waiting for the data service invocations to complete (i.e., *pipelined parallelism*). Moreover, these process instances share between them the set of current top-k results *current-k*.

Algorithm 1: Composition Plan Transformation

Input: C_Q: Composition plan of a Top-k Query Q.
Output: C_Q: Composition plan after transformation.

1 **foreach** *Node* $N_i \in C_Q$ **do**
2 **if** *(N_i is a parallel join operator)* **then**
   `// getting the parallel services`
3 *parallel_services* $\longleftarrow$ N_i.*parents*;
4 $N_j \longleftarrow$ *The_Parent_Of(parallel_services)*;
   `// generating the sub-plans`
5 *Sub_plans* $\longleftarrow$ Generating_BindJoin_SubPlans(*parallel_services*);
   `// applying the transformations on the plan`
6 Removing *parallel_services* from C_Q;
7 Transforming the *parallel join* node N_i to an union node N_i^{union};
8 **for** *each sub plan* $sp \in$ *Sub_plans* **do**
9 Adding an edge from N_j to *root_service*(sp);
10 Adding an edge from *leaf_service*(sp) to N_i^{union};

Algorithm 2: TQEA:Top-k Query Execution Algorithm

Input:
C_Q: Composition plan of a Top-k Query Q; k: answers size; $f(p_i,...,p_j)$: Ranking scoring function;
Output: k best results.

1 *current-k* $\longleftarrow \emptyset$;
   `// Applying the Algorithm 1 to generate the enhanced composition plan`
2 Composition_Plan_Transformation (C_Q) ;
3 **foreach** *Node* $N_i \in C_Q$ **do**
4 **if** *(N_i is a data service s_i)* **then**
5 **if** *(s_i does not have a predecessor)* **then**
   `// Performing a Sorted Invocation over` s_i
6 Launch Root_Service_Instance(s_i) asynchronously;
7 **else** `//` s_i `has a single parent` N_j
   `// Performing a Random Invocation over` s_i
8 Launch Bind_Join_Instance(s_i, N_j) asynchronously;
9 **else**
10 **if** *(N_i is an union node)* **then**
11 Launch Union_Instance(N_i) asynchronously;
12 **else**
13 Launch Output_Instance(N_i) asynchronously;

[5] The invocation over a data service is executed in the server-side, in which it is deployed, i.e., all involved services may invoked and executed in a pipeline-parallel manner.

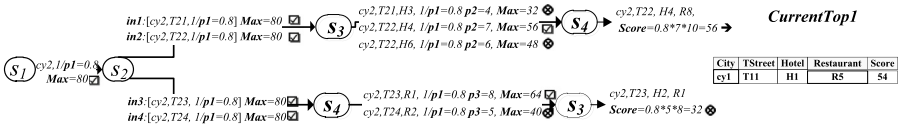


Fig. 6. Execution steps without parallel join operator.

The coordination between all process instances is ensured by the *TQEA* algorithm that schedules the different tasks and data flows, such that, the output of a data service/node is returned (through its process instance) to *TQEA* and then serves as input to another process instance to launch the invocation of its corresponding service/node, etc., until producing the final top-k answers.

When s_i has no parents in C_Q, a *Root_Service_Instance* process is launched (line 6 Algorithm 2). This process manages the invocation of the service s_i by using the *sorted invocation* access method (line 3). In addition, This process checks the stop condition *SC* after each invocation (line 4 and 6) such that, if *SC* holds then the invocation over all process instances is stopped (line 7) and the final top-k results is returned (line 8).

Process: Root_Service_Instance(s_i)

1 $SC \longleftarrow$ **false** // The stop condition
2 **repeat**
3 $t^{out} \longleftarrow$ Sorted_Invocation(s_i);
// Applying the Necessary Invocation Principle
4 $SC \longleftarrow \bar{f}(t^{out}) <$ min(*current-k*) **and** |*current-k*|==k;
5 return t^{out} to the successor nodes ;
6 **until** *SC*;
7 Stopping the composition & shut-downing all process instances;
8 return *current-k* as final results.

In the case when s_i has a predecessor parent N_j[6] (i.e., there is an invocation dependency $N_j \prec s_i$), a *Bind_Join_Instance* process is launched (line 8 Algorithm 2). This process takes input tuples *TInputs* from the output of the parent node N_j in an asynchronous way (line 1). The service s_i is randomly invoked (line 4) with each input tuple $t^{in} \in$ *TInputs*, if the necessary invocation test holds (line 3). The obtained output tuples *TOutputs* are submitted to the successor nodes (line 5).

Process: Bind_Join_Instance(s_i, N_j)

1 **while** *Input tuples TInputs are available from the outputs of* N_j **do**
2 **foreach** *input* $t^{in} \in$ *TInputs* **do**
// Applying the Necessary Invocation Principle
3 **if** $(\bar{f}(t^{in}) \geq$ *min(current-k)* **Or** |*current-k*|$<k$ **)** **then**
4 *TOutputs* $\longleftarrow$ Random_Invocation(s_i, t^{in});
5 return *TOutputs* to the successor nodes;

In the special case when N_i is an union node (obtained after the transformation of a join operator), the TQEA algorithm launches an *Union_Instance* process (line 11 Algorithm 2) that groups the output tuples returned by its predecessors (i.e., from the leaf services in the different sub-plans) and submits them to the next node without applying any filters (line 1–2).

Process: Union_Instance(N_i)

1 **while** *Input tuples TInputs are available from the* N_i*'s predecessors* **do**
2 return *TInputs* to the successor nodes;

The final top-k results is generated by the *Output_Instance* process (line 13 Algorithm 2) that takes input from the output of all nodes that are leaves in the composition plan C_Q (line 1). Thus, for each final output tuple that is fully invoked, the *Output_Instance* process computes its exact query score and checks if it can be added to the *current-k* set (line 2–4).

Process: Output_Instance(N_i)

1 **while** *Input tuples TInputs are available from the* N_i*'s predecessors* **do**
2 **foreach** *input* $t^{in} \in$ *TInputs* **do**
3 **if** $(f(t^{in}) \geq$ *min(current-k)* **Or** |*current-k*|$<k$ **)** **then**
4 Adding t^{in} to *current-k*;

4. Evaluation

4.1. Analytical analysis

In this section, we study the complexity of our top-k algorithm by analyzing the complexity of its steps. The *TQEA* algorithm has two major steps. In the first step, the composition plan C_i is transformed as described by Algorithm 1. Assume J is the number of *parallel join* operators in C_i, and S_{max} is the maximal number of parallel data services per *parallel join* operator, the complexity of the first step is $O(J \times S_{max})$.

In the second step, the composition plan C_i is executed by using a pipeline-parallel style and a multi-processing implementation. This step launches for each service/node a process instance in an asynchronous way (i.e., in parallel). The complexities of launched instances depend on their types and are described, per type, as follows:

- *Root_Instance*: Assume V_{root} is the maximal number of *Sorted Invocations* over each *Root_Instance*, and S_{root} is the number of data services whose invocations are managed by a *Root_Instance* (i.e., they do not have a predecessor node). The complexity of this process instance is $O(V_{root} \times S_{root})$.
- *Bind_Join_Instance*: Assume T_{max} is the maximal number of tuples that are used as inputs to perform *Random Invocations* over a data service, and S_{bind} is the number of data services whose invocations are managed by a *Bind_Join_Instance* (i.e., they have a predecessor node). The complexity of the this process instance is $O(T_{max} \times S_{bind})$.
- *Union_Instance*: Assume T_{max} is the maximal number of tuples that are routed through a union operator, and N_{union} is the number of union operators. The complexity of the this process instance is $O(T_{max} \times N_{union})$.
- *Output_Instance*: Assume T_{final} is the number of final tuples. The complexity of the this process instance is $O(T_{final})$.

[6] This predecessor parent can be either a service s_j or an union operator N^{union}, consequently we use the term parent node N_j to represent the general case.

In summary, the different phases of our algorithm have a polynomial complexity. Consequently, our approach can find the set of top-k answers in a reasonable time and optimal cost, regardless of the composition plan.

4.2. Experiments

In this section we report the experiments we conducted to evaluate the efficiency of our proposed strategies. In our experiments, we tried to answer the following two questions: Q_1: *Can our algorithm answer top-k queries in a reasonable time?*. Q_2: *Can our optimizations reduce the number of unnecessary invocations?*. We evaluated the performance of the proposed approach (referred to as *TQEA*) by comparing it with the following algorithms:

- *Pipeline*: This algorithm retrieves the set of the top-k answers by using the *pipeline parallel strategy*, such that the necessary invocation principle is applied only on the root service in order to stop the composition running.
- *Single-Plan*: This algorithm is similar to our proposed algorithm in that both strategies are applied. However, *parallel join* operators are not transformed and all input tuples are routed through a single plan.

These algorithms are compared by measuring two different performance metrics: the total running time to retrieve the k best results and the overall number of avoided invocations. We repeated our experiments as we varied the values of our various parameters including: the number of required answers k, the average number of output tuples of random services and the random invocation cost.

Our algorithm is implemented in Java. The experiments were conducted on a 1.8 GHz Intel i7 8th Gen core CPU and 8 GB of RAM, running Windows. Our data services are implemented using the MicroService Architecture (see Section 3.1), and are deployed independently in different physical machines and Cloud VMs (see Fig. 3). In our implementation, each data service instance is deployed in an autonomous Virtual Machine that has 4 cores running at 2.6 GHz and with 3 GB of RAM, each.

4.3. Performance versus the number of k best results

In this experiment, we measured the performance improvement introduced by our proposed strategies while varying the number of k best results from 10 to 100 and fixing the average number of output tuples at 20 *tuples* and the random invocation cost at 0.5. Fig. 7(a) compares the total running time required by the algorithms described above to evaluate a top-k query. In contrast to *Pipeline* that requires a significant running time, *TQEA* and *Single-Plan* show a faster processing with an obvious domination of *TQEA*. The long running time required by the *Pipeline* algorithm is due to the fact that it does not filter out unnecessary invocations of intermediate data services. Fig. 7(b) compares the number of avoided invocations. In contrast to *Pipeline* which avoids only one invocation (i.e., requires a complete execution) regardless of the required value of k, *TQEA* and *Single-Plan* minimize the evaluated tuples by increasing the number of avoided invocations. For smaller values of k (between 10 and 25), both *TQEA* and *Single-Plan* algorithms eliminate almost the same number of unnecessary invocations. However, for large values of k, *TQEA* significantly outperforms (the overhead is up to +15%) *Single-Plan* by avoiding more invocations, especially those that are performed through *parallel join* operators. This difference between *TQEA* and *Single-Plan* did not appear in the previous experiment while measuring the total running time (Fig. 7(a)). This is explained by the fact that we have used small values for the fixed parameters: *average number of output tuples* and *random invocation cost* (20 and 0.5, respectively), such that the added unnecessary invocations in *Single-Plan* over *parallel join* does not increase significantly the evaluation time.

4.4. Performance versus the average number of output tuples

In this experiment, we fixed the value of k at 40 and the random invocation cost at 0.5 and varied the average number of output tuples from 5 to 40. Fig. 8(a) reports the relative processing time. When the average number of output tuples is less or equal to 20, both *TQEA* and *Single-Plan* generate 40 ranked results in less than 4 s; but when that number increases, *TQEA* performs better than *Single-Plan*, such that the difference in performance increases in a portion between 40% to 60%. Similar observations can be made for the second experiment which compares the number of avoided invocations (see Fig. 8(b)). The obtained results show that the number of avoided invocations is largely influenced by the average number of output tuples, such that, when that number increases, the number of intermediate tuples increases and consequently the number of unnecessary random invocations. For large values of the average number of output tuples (greater or equal to 30 *tuples*), *TQEA* algorithm dominates *Single-Plan* algorithm (the overhead is up to +40%) by eliminating more unnecessary random invocations, which significantly improves and enhances the evaluation of top-k queries over data services.

4.5. Performance versus the random invocation cost

In this experiment, we evaluated only the total running time by varying the random invocation cost. The avoided invocations metric was not taken into account in this experiment since it is clear that the random invocation cost does not have any influence on the necessary invocation principle. Fig. 9(a) compares the running time by varying the random invocation cost from 0.01 to 1 and fixing k to 40 and the average number of output tuples to 10. For small values of the random invocation cost (between 0.01 and 0.1), all algorithms perform well, where the final results are obtained in less than 1 s. This is explained by the fact that the not-eliminated invocations in the *Pipeline* algorithm do not increase the processing time, since their cost is a very low. In the second evaluation (Fig. 9(b)), we increased the average number of output tuples to 40 (rather than 10) and compared the total running times. The obtained results show that the *Single-Plan* algorithm has the best performance for low cost values and *TQEA* has the best performance for high cost values. The reason is that when there is high number of intermediated tuples with a very low random invocation cost the best way is to route all intermediate tuples through the same plan without the need to transform the *parallel join* operators into a set of sub-plans.

5. Related work

5.1. Data web service composition

The authors in [3] proposed a query optimization approach for SPJ queries (Select-Project-Join) involving data services. Specifically, they proposed a greedy algorithm that uses the bottleneck cost metric to generate the optimal composition plan. The bottleneck cost is derived based on a parallel execution model.

The authors in [23] tackled the issue of optimizing SPJ queries that access the data market through RESTful data services. Since access to data in the data market may not be free, the authors proposed PayLess, a system that helps data buyers to optimize their queries so that they can obtain the query results by paying

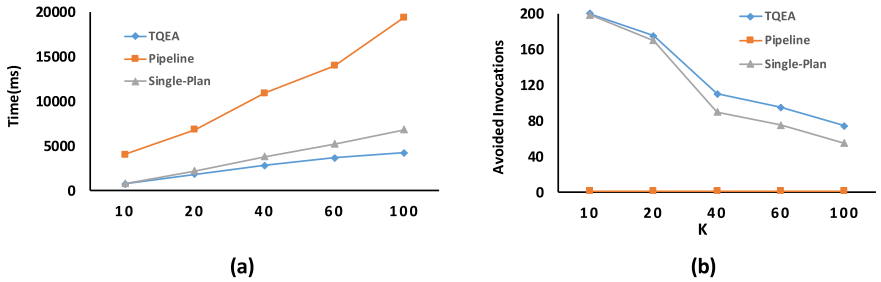


Fig. 7. The effect of k on TQEA, Pipeline and Single-Plan.

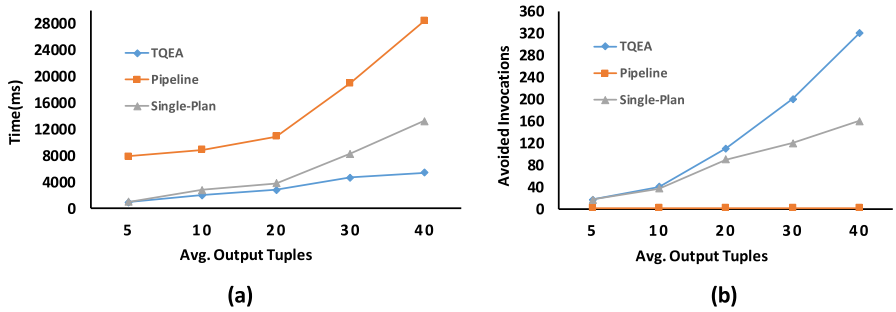


Fig. 8. The effect of output tuples on TQEA, Pipeline and Single-Plan.

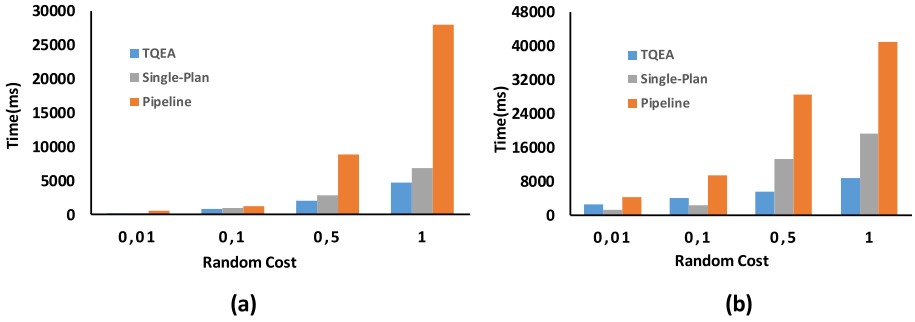


Fig. 9. The effect of random invocation cost on TQEA, Pipeline and Single-Plan.

less to the data sellers. Their approach aims to generate the best composition plan that reduces the amount of intermediate data, and thus the number of queries in the purchasing process.

The authors in [2] proposed an approach to automatically compose data services with uncertain semantics. They proposed a probabilistic approach, in which the uncertain semantics of a data service is represented by several possible semantic views, each one is associated with a probability. They also define an efficient algorithm to evaluate queries over data services under uncertain semantics. The work [12] has further extended that approach by taking into account the correlation relationships between possible semantics views.

An interesting work [13] has addressed the composition of WoT services (i.e., Web of Things data services), such that the composition of connected objects (things) will become a composition of WoT services. The data returned by WoT services are often uncertain due to various reasons. Authors in that work propose a probabilistic solution to model and evaluate queries over WoT services while handling the data uncertainty problem.

However, none of these works have addressed the ranking problem over data services which is the main focus of this paper. In our work, we propose a service composition approach that evaluates top-k queries over free data services. In addition, the data exchanged between services are supposed certain.

5.2. Top-k query optimization

Top-k queries processing and optimization is a topic that has been studied in many areas : relational databases, uncertain data, RDF data, location-based services, data graph [22]. In this paper, we review the works that focus on answering Top-k query over data web services.

The authors in [24] proposed a service composition approach that optimized multi-domain queries on the Web (i.e., over heterogeneous Web information sources). Two kinds of data services are considered: *search service* and *exact service*, where the former is used to formulate the top-k queries. However, the proposed optimization heuristics and cost metrics are more suitable for SPJ queries than for top-k queries. Contrary to this work, our proposed strategies and algorithms are designed to efficiently evaluate top-k queries over data services by ignoring SPJ query characteristics that increase the amounts of the executed unnecessary invocations, and consequently the evaluation cost (i.e., we do not need to fetch and generate all possible results).

The authors in [15] proposed a pulling strategy, called *Cost-Aware with Random and Sorted access* (CARS), that optimizes the retrieval of the top-k combinations that can be formed by joining data services. This pulling approach defines the optimal execution schedule (i.e., service composition) which determines the order of invocation of the joined data services while controlling the

number of random access. The execution schedule is obtained based on an optimization problem that uses some statistical parameters characterizing the joined services. The proposed cost model considers data services that must expose both sorted access and random access. However, this approach considers only top-k queries that involve at most two data services. In addition, these services must be available with both sorted access and random access. In other words, the proposed solution focuses on a straightforward class of top-k queries, in which there are no invocation dependencies between data services. The work in [25] introduces an extension of [15] by considering only data services with no-random access. Similarly to these works, our necessary invocation principle aims to reduce the number of random invocations. However, we take into account more complex top-k query plans that involve invocation dependencies between data services, i.e., some services are available with only random access. In addition, we adopt a pipelined execution model in order to reduce the unnecessarily long service composition processing time. Finally, the approaches [15,25] are not designed to handle real-life top-k queries contrary to our work.

The authors in [26] introduce an approximate top-k algorithm for web data services. This work extends the PBRJ framework [27] with a new probabilistic component. However, in our work we proposed an exact ranking algorithm that minimizes the access to unnecessary invocations.

There are also other top-k queries approaches [16,28–31] that propose solutions for several data service challenges, such as: top-k Cloud Services Query, Query Reranking, web services recommendation, data service privacy, etc.

6. Conclusion

Stimulated by the importance of ranking queries over external data sources, in this paper we presented an approach that efficiently evaluates top-k queries over data services. Specially, we adopted a pipelined parallelism query execution model that follows a non-additive cost model. Then, we defined the necessary invocation principle that aims to avoid unnecessary random invocations that require an expensive cost. We evaluated thoroughly our approach, the obtained results proved its efficiency.

Our approach has some limitations described as follows:

- In this paper, we are particularly interested in parallel join operators that take into account only random services. However, in some top-k queries, we can have also parallel join operators that involve sorted data services.
- In this paper, a top-k query involves only data services. However, some queries may be formulated with both data services and classical relational data sources.
- In our approach, the ranking function of the top-k query must be monotone. However, different models are available: Unspecified ranking function, Generic ranking function, etc.

As a future work, we intend to extend our approach to handle the above limitations. In addition, we plan to address the data uncertainty, i.e., how to evaluate a top-k query over data services whose returned data are uncertain.

CRediT authorship contribution statement

Abdelhamid Malki: Conceptualization, Methodology, Writing - original draft, Software. **Sidi-Mohamed Benslimane:** Conceptualization, Resources, Supervision. **Mimoun Malki:** Project administration, Resources, Validation. **Mahmoud Barhamgi:** Conceptualization, Writing - review & editing. **Djamal Benslimane:** Conceptualization, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

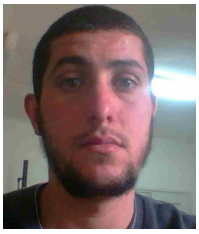
Acknowledgment

This research was supported by the Algerian Directorate General for Scientific Research and Technological Development, Algeria under grant No. 01/ESI Sidi Bel Abbes/DGRSDT/2019.

References

[1] C. Li, K.C.-C. Chang, I.F. Ilyas, S. Song, Ranksql: Query algebra and optimization for relational top-k queries, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, in: SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 131–142.
[2] A. Malki, M. Barhamgi, S. Benslimane, D. Benslimane, M. Malki, Composing data services with uncertain semantics, IEEE Trans. Knowl. Data Eng. 27 (4) (2015) 936–949.
[3] U. Srivastava, K. Munagala, J. Widom, R. Motwani, Query optimization over web services, in: VLDB, 2006, pp. 355–366.
[4] A.Y. Levy, A. Rajaraman, J.D. Ullman, Answering queries using limited external query processors, J. Comput. System Sci. 58 (1) (1999) 69–82.
[5] S. Hwang, K.C. Chang, Probe minimization by schedule optimization: Supporting top-k queries with expensive predicates, IEEE Trans. Knowl. Data Eng. 19 (5) (2007) 646–662.
[6] G. Zhang, K. Ren, J. Ahn, S. Ben-Romdhane, Grit: Consistent distributed transactions across polyglot microservices with multiple databases, in: 2019 IEEE 35th International Conference on Data Engineering (ICDE), 2019, pp. 2024–2027.
[7] T. Kiss, P. Kacsuk, J. Kovacs, B. Rakoczi, A. Hajnal, A. Farkas, G. Gesmier, G. Terstyanszky, Micado—microservice-based cloud application-level dynamic orchestrator, Future Gener. Comput. Syst. 94 (2019) 937–946.
[8] R. Yu, V.T. Kilari, G. Xue, D. Yang, Load balancing for interdependent iot microservices, in: IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, 2019, pp. 298–306.
[9] M. Barhamgi, D. Benslimane, B. Medjahed, A query rewriting approach for web service composition, IEEE Trans. Serv. Comput. 3 (3) (2010) 206–222.
[10] M. Barhamgi, C. Ghedira, D. Benslimane, S. Tbahriti, M. Mrissa, Optimizing daas web service based data mashups, in: IEEE International Conference on Services Computing, SCC 2011, Washington, DC, USA, 4-9 July, 2011, 2011, pp. 464–471.
[11] S. Amdouni, D. Benslimane, M. Barhamgi, A. HadjAli, R. Faiz, P. Ghodous, A preference-aware query model for data web services, in: Conceptual Modeling - 31st International Conference ER 2012, Florence, Italy, October 15-18, 2012. Proceedings, 2012, pp. 409–422.
[12] A. Malki, D. Benslimane, S.-M. Benslimane, M. Barhamgi, M. Malki, P. Ghodous, K. Drira, Data services with uncertain and correlated semantics, World Wide Web 19 (1) (2016) 157–175.
[13] S. Awad, A. Malki, M. Malki, M. Barhamgi, D. Benslimane, Composing wot services with uncertain data, Future Gener. Comput. Syst. 101 (2019) 940–950.
[14] N. Bruno, L. Gravano, A. Marian, Evaluating top-k queries over Web-accessible databases, in: Proceedings 18th International Conference on Data Engineering, 2002, pp. 369–380.
[15] D. Martinenghi, M. Tagliasacchi, Cost-aware rank join with random and sorted access, IEEE Trans. Knowl. Data Eng. 24 (12) (2012) 2143–2155.
[16] A. Asudeh, N. Zhang, G. Das, Query reranking as a service, Proc. VLDB Endow. 9 (11) (2016) 888–899.
[17] I.F. Ilyas, W.G. Aref, A.K. Elmagarmid, Supporting top-k join queries in relational databases, VLDB J. 13 (3) (2004) 207–221.
[18] A. Shanbhag, H. Pirk, S. Madden, Efficient top-k query processing on massively parallel hardware, in: Proceedings of the 2018 International Conference on Management of Data, in: SIGMOD '18, ACM, New York, NY, USA, 2018, pp. 1557–1570.
[19] A. Marian, N. Bruno, L. Gravano, Evaluating top-k queries over web-accessible databases, ACM Trans. Database Syst. 29 (2) (2004) 319–362.
[20] A. Deshpande, L. Hellerstein, Parallel pipelined filter ordering with precedence constraints, ACM Trans. Algorithms 8 (4) (2012) 41:1–41:38.
[21] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, J. Widom, Adaptive ordering of pipelined stream filters, in: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, in: SIGMOD '04, ACM, New York, NY, USA, 2004, pp. 407–418.

[22] I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Comput. Surv. 40 (4) (2008) 11:1–11:58.
[23] Y. Li, E. Lo, M.L. Yiu, W. Xu, Query optimization over cloud data market, in: Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015., 2015, pp. 229–240.
[24] D. Braga, S. Ceri, F. Daniel, D. Martinenghi, Optimization of multi-domain queries on the web, PVLDB 1 (1) (2008) 562–573.
[25] A. Abid, M. Tagliasacchi, Provisional reporting for rank joins, J. Intell. Inf. Syst. 40 (3) (2013) 479–500.
[26] A. Wagner, V. Bicer, T. Tran, Pay-as-you-go approximate join top-k processing for the web of data, in: V. Presutti, C. d'Amato, F. Gandon, M. d'Aquin, S. Staab, A. Tordai (Eds.), The Semantic Web: Trends and Challenges, Springer International Publishing, Cham, 2014, pp. 130–145.
[27] K. Schnaitter, N. Polyzotis, Evaluating rank joins with optimal cost, in: Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, in: PODS '08, ACM, New York, NY, USA, 2008, pp. 43–52.
[28] K. Benouaret, I. Benouaret, M. Barhamgi, D. Benslimane, Efficient top-k cloud services query processing using trust and qos, in: S. Hartmann, H. Ma, A. Hameurlain, G. Pernul, R.R. Wagner (Eds.), Database and Expert Systems Applications, Springer International Publishing, Cham, 2018, pp. 203–217.
[29] L. Ren, W. Wang, An svm-based collaborative filtering approach for top-n web services recommendation, Future Gener. Comput. Syst. 78 (2018) 531–543.
[30] Y. Zheng, R. Lu, X. Yang, J. Shao, Achieving efficient and privacy-preserving top-k query over vertically distributed data sources, in: ICC 2019 - 2019 IEEE International Conference on Communications (ICC), 2019, pp. 1–6.
[31] X. Liu, R. Lu, J. Ma, L. Chen, B. Qin, Privacy-preserving patient-centric clinical decision support system on Naïve Bayesian classification, IEEE J. Biomed. Health Inf. 20 (2) (2016) 655–668.



Abdelhamid Malki received the Master degree in computer science from the University of Sidi Bel Abbes (Algeria), in 2011 and the PHD degree in computer science from Claude Bernard Lyon 1 University(France), in 2015. Currently, he is associated professor at the higher School of Computer Science, Sidi Bel-Abbès, Algeria. His current research interests lie in the area of WOT, service computing, Data integration, Top-k queries optimization and probabilistic databases.

Sidi-Mohamed Benslimane is a full Professor at the higher School of Computer Science, Sidi BelAbbès, Algeria. He is currently Head of Research Team 'Service Oriented Computng' at the LabRiSBA Laboratory. His research interests include, semantic web, ontology engineering, distributed and heterogeneous information systems and cloud computing.

Mimoun Malki He is a full Professor at the higher School of Computer Science, Sidi Bel-Abbès, Algeria. He is the head of LabRi-SBA Laboratory. His research interests include Databases, Information Systems Interoperability, Ontology Engineering, Linked Data, Semantic Web Services, Enterprise Mashup, WOT and Cloud Computng.

Mahmoud Barhamgi is an Associate Professor of computer sciences at Claude Bernard Lyon 1 University. He is a member of the SOC research team of the Lyon Research Center for Images and Intelligent Information Systems associated to the French National Center for Scientific Research (LIRIS-CNRS) in Lyon, France. His research interests include Data Integration, Mashups and Service Oriented Computng.

Djamal Benslimane is a full professor of computer sciences at Lyon 1 University and Co-head of Service Oriented Computng (SOC) research team of the Lyon Research Center for Images and Information Systems in Lyon, France. His research interests include Distributed Information Systems, Web services, Ontologies and Databases. He has published papers in well-known journals (ACM TOIT, SIGMOD Record, IEEE TSC, IEEE IC, etc.).