World Wide Web (2012) 15:1–31
DOI 10.1007/s11280-011-0121-9

Multi-attribute optimization in service selection

Qi Yu · Athman Bouguettaya

Received: 12 April 2010 / Revised: 3 February 2011 /
Accepted: 4 February 2011 / Published online: 25 February 2011

Abstract As multiple service providers may compete to offer the same functionality with different quality of service (e.g., latency, fee, and reputation), a key issue in service computing is selecting service providers with the best user desired quality. Existing service selection approaches mostly rely on computing a predefined objective function. When multiple quality criteria are considered, users are required to express their preference over different (and sometimes conflicting) quality attributes as numeric *weights*. This is a rather demanding task and an imprecise specification of the weights could miss user desired services. We propose a multi-attribute optimization approach to tackle this issue. In particular, we develop a novel concept, called *service skyline*, and a set of *service skyline computation* techniques that return a set of *most interesting* service providers. These providers are non-dominant in all user interested quality attributes. Thus, the service skyline ensures that the user desired providers will be included. Analytical and experimental studies justify the performance of the proposed techniques. The relative small sizes of the service skylines also make it practical for service users to make selections from them.

Keywords Web service · quality of service · service selection

1 Introduction

Service oriented computing is emerging as the preferred platform for deploying new applications [23, 31]. Industry leaders like IBM, Microsoft, Yahoo, Google,

Q. Yu (✉)
College of Computing and Information Sciences, Rochester Institute of Technology,
Rochester, NY, USA
e-mail: qi.yu@rit.edu

A. Bouguettaya
School of Computer Science and Information Technology, RMIT,
Melbourne, Victoria, Australia
e-mail: athman.bouguettaya@rmit.edu.au

HP, and others are strongly behind this push. There is strong evidence that if the right Web service infrastructure is in place, this will spur entrepreneurship in deploying novel Web services that will compete to provide differentiated services (www.programmableweb.com, [10, 26]). There is also a strong impetus by IBM to define the field of services science, where Web services will play a major deployment role (www.research.ibm.com/ssme/). Therefore, it is realistic to expect that Web services will increase many fold in the future [4]. The growing number of Web services gives users more options because multiple service providers may compete to offer the same functionality. However, it also brings users another problem: selecting a proper provider with the desired quality of service. Typically, users have to go through a series of trial-run processes. It would be even more painstaking if users want to target the providers that *best* suit their preference. Therefore, the user may want to include the quality requirement into the search criteria. In this case, it is necessary to differentiate competing Web services based on user expected *Quality of Web Service (QoWS)*.

Existing service optimization approaches usually select services based on a predefined objective function [19, 29, 30, 33, 35]. They require users to express their preference over different (and sometimes conflicting) quality parameters as numeric *weights*. The objective function assigns a scalar value to each service provider based on the quality values and the weights given by the service user. The provider gaining the highest value from the objective function will be selected and returned to the user. Implementing such an optimization strategy may pose several challenges:

- Transforming personal preferences to numeric weights is a rather demanding task for users. Sometimes it is even impossible if the preference is still vague before the user is presented with the actual service providers. Users may miss their desired providers because of an imprecise specification of the weights, which would be very common in real-world scenarios.
- Users may lose the flexibility to select their desired providers by themselves. For example, a service user may choose a service provider that has a good reputation within a price range she can tolerate although price is a very important factor she considers. In this case, the relationship between reputation and price is subtle and the choice from different users may vary significantly. Therefore, it would be wise to give users the flexibility to make their own selections from a small set of candidate providers.

We propose a novel concept, called service skyline*, that tackles the service selection problem from a perspective which is completely different from all existing service selection approaches. Computing a service skyline guarantees to include the best user desired service providers without any user intervention.* Skyline computation has recently received considerable attention in database community [5, 15, 20, 27]. For a d-dimensional data set, the skyline consists of a set of points which are not dominated by any other points. A point $\vec{p}\ (p_1, \ldots, p_d)$ dominates another point $\vec{r}\ (r_1, \ldots, r_d)$ if $\forall\ i \in [1, d]$, $p_i \succeq r_i$ and $\exists\ j \in [1, d]$, $p_j \succ r_j$. We use $\succeq$ to generally represent *better than or equal to* and $\succ$ to represent *better than*. In the context of Web services, a service skyline can be regarded as a set of service providers or their compositions that are not dominated by others in terms of all user interested QoWS attributes, such as response time, fee, and reputation. A formal definition about the service

skyline will be given in Section 2. Computing service skylines brings two key benefits for service selection:

- Service skylines are computed automatically based on the inherent QoWS features of service providers. Thus, it completely frees service users from the challenging weight assignment task.
- Computing service skylines won't lose any merit of using the objective function. This is due to a major property of the skyline. For a set $\mathcal{S}$ and any monotone objective function $\mathcal{S} \rightarrow \mathbb{R}$, if $\vec{r} \in \mathcal{S}$ maximizes the objective function, then $\vec{r}$ is in the skyline [5]. Thus, no matter how the weights are assigned, the skyline guarantees that the user desired service providers are included so that users can make flexible selection from them. In addition, the users can always choose to use any monotone objective function they prefer after the skyline is computed. The optimal solution will always be the same as computed from the original service space but with a much efficient manner because of the much smaller skyline size.

The service skyline algorithms proposed in this paper are developed based upon a foundational service framework presented in [30]. A service model is provided by this framework that defines a service schema and a service relation. The service schema captures the key features of Web services across an application domain and the service relation is used to store QoWS information of service providers. The major contributions of this paper are summarized as follows:

- We formally define the concept of service skyline. We identify the key differences between service skylines and database skylines and point out the challenges for computing service skylines.
- We investigate how to leverage the indices on service operations to compute the service skylines. This study helps identify some inherent issues, which lead us to develop more efficient service skyline algorithms.
- We present two service skyline algorithms. The Baseline Algorithm (BA) adopts a two-level pruning scheme to efficiently compute the service skyline. The second algorithm, called OGI, is built upon a novel indexing structure.
- As a further refinement, we propose a hybrid indexing structure that combines a R-tree with a partition tree. The hybrid structure optimizes both dominance checking (using the R-tree) and incompatibility checking (using the partition tree) to achieve good performance and scalability.
- We analytically and experimentally evaluate the proposed service skyline algorithms in terms of both the performance and the sizes of the service skylines. The results show that the algorithms are efficient and the sizes of the service skylines are small and stable. These justify that the proposed approach provide an promising solution for service selection.

Example 1.1 As a running example, we use an application from the car brokerage domain. We consider a customer, say Mary, planning to buy a used car having a specific model, make, and mileage. Assume that Mary has access to a Web service infrastructure where the different entities that play a role in the car purchase are represented by Web services. Typical Web services that need to be accessed include car purchase, car insurance, and financing services. A single Web service may provide multiple operations that have dependency relationships. We also anticipate that

there will be multiple competing service providers with different QoWS (e.g., the fee they charge, their reputation, etc). To purchase an entire car package, Mary would first like to know the price quote of the selected car and the vehicle history report. She then needs to get the insurance quote. Finally, since Mary needs the financing assistance, she also wants to know the financing quote. Essentially, Mary wants to get an entire package with low price and from creditable (i.e., with good reputation) service providers.

The remainder of this paper is organized as follows. We formally define the service skyline problem in Section 2. We investigate how to leverage the indices on service operations to compute service skylines in Section 3. We present two service skyline algorithms in Section 4 and provide a cost model for performance study and improvement. We experimentally evaluate and compare the proposed algorithms under different settings in Section 5. We discuss related work in Section 6 and provide some concluding remarks in Section 7.

2 Preliminaries

In this section, we start with a brief introduction of the foundational service framework presented in [30] because the service skyline algorithms are developed based on this framework. We then formally define the service skyline problem.

2.1 The service framework

The service framework is built around a formal service model, which provides foundational support for service query and service selection. The design of this model is inspired by the standard relational model and makes some key extensions from it. The service model captures a set of essential semantics of Web services, including functionality, dependencies, and quality, which are all of primary interest for users to access services. The service model defines two important concepts: *service schema* and *service model*.

Definition 2.1 (Service schema [30]) A service schema $\mathcal{S}$ is defined as a tuple $(SG_1, \ldots, SG_n, \mathcal{D})$, where each SG_i is a DAG, called *service graph*. In $SG_i = (V_i, E_i, \epsilon_i)$, the vertex set V_i represents the set of service operations in the service graph, the edge set E_i represents the dependency constraints between service operations, and ϵ_i is the root of the service graph representing the entry point, through which all other operations in the service graph can be accessed. $\mathcal{D}$ represents the set of dependencies between two non-root operations from different service graphs.

Figure 1 shows the service schema for Example 1.1. The service schema contains three service graphs, representing the *Car Purchase (CP)*, *Car Insurance (CI)*, and *FInancing (FI)* services. For example, in *CI*, there are a set of service operations, such as `drivingHistory` and `insuranceQuote`. These operations collectively represent the functionalities of the *CI* Web service. The dependencies between service operations are captured by the edges in the service graph. For example, (`drivingHistory`, `insuranceQuote`) means that the execution of

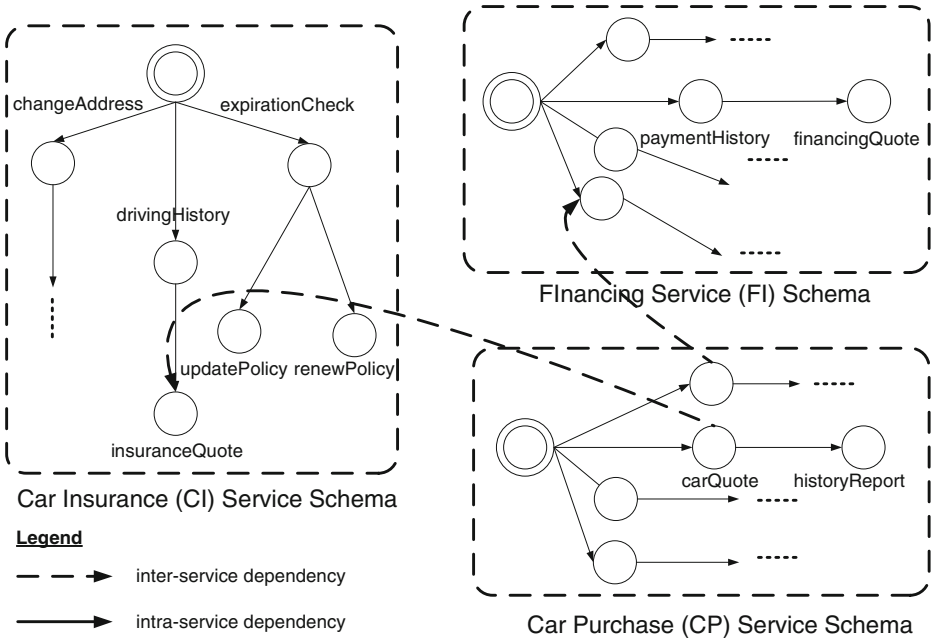

Figure 1 The service schema for car brokerage.

`insuranceQuote` depends on the result of `drivingHistory`. Service operations from different Web services could have an inter-service dependency. For example, there is a dependency between `carQuote` and `insuranceQuote`. It is denoted as (`carQuote`, `insuranceQuote`).

The dependencies in the service schema determine the invocation sequence of service operations. Since an operation can only be invoked after the invocation of all its dependent operations, a key concept called *operation graph* is introduced to capture these operations and the dependencies between them.

Definition 2.2 (Operation graph [30]) For a service graph $SG = (V, E, \epsilon)$, an operation graph $G(op)$ is the *union* of all the paths in SG that lead to operation op. $G(op)$ is a *subgraph* of the service graph SG. Figure 2 shows an operation graph $G(d)$, which

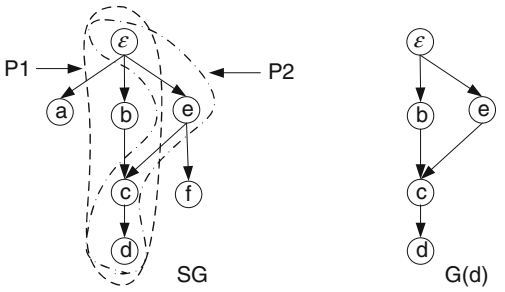

Figure 2 An example of an operation graph.

is formed from SG by the union of two paths, P_1 and P_2, that both lead to the service operation d.

Definition 2.3 (Operation set graph [30]) For a service graph $SG = (V, E, \epsilon)$, we define an operation set graph $G(\mathbf{op}) = \cup_{i=1}^{k} G(op_i)$, where $\mathbf{op} = \{op_i | 1 \leq i \leq k\}$. $G(\mathbf{op})$ is a *subgraph* of service graph SG. For example, in Figure 2, the operation set graph for {a,d,f} is SG itself, i.e., $G(\{a, d, f\}) = SG$.

Given a (set of) service operation(s) that a service user wants to access and a service graph, an operation (set) graph can be directly obtained through standard graph algorithms. The operation (set) graph includes all operations that are necessary for the user's request. It also captures the dependencies between these operations, which determine their invocation order. When a user wants to access service operations from multiple service graphs, the corresponding service graphs can be composed on demand. The composed graph, $G' = G_i \circ G_j$, is formed by coalescing the root of G_i and G_j. The inter-graph edges become part of the edge set V' in the newly formed service graph G'. The newly generated root needs to store the entry information (e.g., URI) for accessing service operations from original service graphs.

The service relation is used to store the quality of different service providers. It defines a set of service instances that conform to the service schema, i.e., the service instances offer the operations and follow the dependency constraints defined in the service graphs. However, since the service instances are provided by different service providers, they may have different quality properties.

Definition 2.4 (Service relation [30]) A service relation SR with a service graph $SG = (V, E, \epsilon)$ is defined as a set of service instances $\mathcal{I} = \{(sid, op_1, \ldots, op_n)\}$, where sid is the unique service id; op is a service operation and defined as a pair $op = (opid, \mathcal{Q}(op))$, where $opid$ is the operation id and $\mathcal{Q}$ is a set of QoWS parameters of op. Typical QoWS parameters include latency, fee, reliability, availability, and reputation. Other parameters can be added based on the application domain.

2.2 Problem definition

Given an operation (set) graph, we can perform a *topological sort* on this graph, which will order the operations based on their dependencies. This operation sequence is referred to as a *generic service plan*. Service Execution Plans (SEPs) can be generated by instantiating the generic service plan with the operations from the service instances in the service relation. Table 1 shows these aggregation functions for some typical QoWS parameters. The quality parameters of a SEP can be computed

Table 1 QoWS for a SEP.

QoWS parameter	Aggregation function
lat(SEP)	$\sum_{i=1}^{n} lat(op_i)$
rel(SEP)	$\sum_{i=1}^{n} \log(rel(op_i))$
$avail$(SEP)	$\sum_{i=1}^{n} \log(avail(op_i))$
fee(SEP)	$\sum_{i=1}^{n} fee(op_i)$
rep(SEP)	$\frac{1}{n} \sum_{i=1}^{n} rep(op_i)$

by aggregating those of its member service operations. These aggregation functions are widely used in dealing with various service optimization problems [30, 33, 35]. Once the quality parameters of the SEPs are available, the quality of a SEP can be represented as a vector, such as (lat, rel, avail, fee, rep).

Definition 2.5 (SEP_A dominates SEP_B) Consider a set of user interested QoWS parameters, $\mathcal{Q}(q_1, \ldots, q_d)$. SEP_A dominates SEP_B (denoted as $SEP_A \rhd SEP_B$) when $\forall\, i \in [1, d], q_i^A \succeq q_i^B$ and $\exists\, j \in [1, d], q_j^A \succ q_j^B$.

Definition 2.6 (Service skyline) A service skyline (a.k.a. SEP skyline) consists of a set of SEPs that are not dominated by other SEPs.

Example 2.1 Consider the following service request: a user wants to get an insurance quote from a car insurance service and she is interested in two quality aspects, service fee and response time. To fulfill this service request, a generic execution plan will be first generated that consists of two operations: (`drivingHistory`, `insuranceQuote`). It is worth to note that `drivingHistory` is dynamically identified based on the dependency. Assume that there are five providers that offer this car insurance service and their corresponding QoWS parameters are shown in Figure 3. Instantiating the above generic execution plan will result in five SEPs: SEP_1, SEP_2, SEP_3, SEP_4, and SEP_5. Based on the aggregation functions defined in Table 1, we can compute the QoWS of these SEPs as: (4, 25), (6, 20), (2, 40), (9, 20), (4.5, 25). Based on Definition 2.5, we have $SEP_1 \rhd SEP_5$ and $SEP_2 \rhd SEP_4$. Thus, the service skyline will consist of three SEPs: SEP_1, SEP_2, and SEP_3.

SEPs are inherently different from data in a database. This poses a set of new challenges for computing a service skyline. For example, SEPs are not stored prior to the submission of a service request. Instead, for each service request, a set of SEPs will be dynamically generated. A SEP usually contains a set of member service operations, some of which may even not appear in the service request. The operations are selected and put in the SEP dynamically based on the dependency constraints. Since the service providers are selected based on user's preference on the entire SEP, the selection can only be carried out after the SEPs have been generated. Furthermore, the attributes of a SEP are aggregates of the corresponding attributes from its member operations. Answering aggregate queries remains to be a challenging task in traditional database systems. It is usually addressed by leveraging

Figure 3 Sample service providers for car insurance.

	DrivingHistory		InsuranceQuote	
sid	Latency	Fee	Latency	Fee
1	2	25	2	0
2	3	15	3	5
3	1	30	1	10
4	5	10	4	10
5	1.5	20	3	5

materialized views [9, 12, 25]. However, view materialization may not be suitable for the highly dynamic SEP spaces.

Summary of contributions We propose two algorithms to efficiently compute service skylines. The first algorithm relies on the indices built from service operations, which are relatively static and can be stored and indexed beforehand. The second algorithm relies on the operation graphs from the service schema to directly index the SEP space. The key contributions are summarized as follows.

- Even though SEPs are dynamically generated for each service request, the service operations are relatively static. Therefore, the QoWS parameters of service operations can be stored and indexed beforehand. We develop a dual-pruning mechanism, which exploits the service operation index to filter out dominated SEPs and then uses an in memory structure to generate the final skyline.
- Services are typically not accessed in an ad hoc manner. Instead, certain business logics usually need to be followed. For example, in the travel domain, if one reserves a package that includes both airline booking and car rental, the pickup time and location of the rental car is automatically determined by the air ticket. The business logics are captured by the service schema in terms of dependency constraints. These constraints only allow SEPs that conform to certain patterns to be generated. We propose to leverage operation graphs, which can be derived from the service schema, to directly build indices for SEPs. We discuss why operation graphs offer sufficient information to index SEPs and present the technical details for constructing operation graphs and how to use them to index the SEP space.
- The indices on service operations and SEPs can be used to effectively prune the dominated SEPs. However, for the SEPs that cannot be pruned by the indices (which include all skyline SEPs), pairwise comparisons are required in order to determine whether they are in the skyline or not. With the increase of the skyline size, pairwise comparisons can become very expensive, which will slow down the entire skyline computation process. Inspired by recent works on skyline computation in the database community [17, 36], we propose a hybrid indexing structure that combines a R-tree with a space partition tree. The space partition tree can effectively reduce the number of pairwise comparisons by dividing the SEPs into different partitions such that no comparisons are needed for SEPs that belong to *incomparable* partitions. In this regard, the proposed hybrid structure optimizes both dominance checking (using the R-tree) and incomparability checking (using the partition tree). Experimental study demonstrated that it achieves good performance and also scales well to high dimensionality.

3 Indexing service operations

In this section, we investigate how to leverage indices on service operations to possibly expedite service skyline computation. The following theorem allows us to compute the service skyline based on the indices of service operations. We define a notation that will be used for explaining the algorithms. Consider two SEPs: SEP_1 and SEP_2. $SEP_1 \triangleright^{op} SEP_2$ represents that SEP_1 dominates SEP_2 on operation *op*. For instance, in Example 2.1, we have $SEP_1 \triangleright^{\text{insuranceQuote}} SEP_5$.

Theorem 3.1 *For any two SEPs, SEP_1 and SEP_2, that contain k service operations $op_1, \ldots, op_k$,*

$$\left(SEP_1 \rhd^{op_1} SEP_2\right) \wedge \ldots \wedge \left(SEP_1 \rhd^{op_k} SEP_2\right)$$
$$\Rightarrow SEP_1 \rhd^{(op_1,\ldots op_k)} SEP_2 \ (i.e., \ SEP_1 \rhd SEP_2)$$

Theorem 3.1 states that if SEP_1 can dominate SEP_2 on all member operations, then SEP_1 dominates SEP_2. Thus, these dominated SEPs can be eliminated from the SEP space. The underlying strategy is to examine dominance relationship between two SEPs on each service operation. If a SEP has all operations dominated by another SEP, the former SEP can be safely removed. Checking the dominance relationship between service operations can be efficiently achieved through the indices on these operations. In what follows, we will focus on extending two algorithms, B-tree based approach and nearest neighbor search approach. The strategy can be easily applied to database skyline algorithms with other types of indices.

3.1 Indexing service operations using B-trees

We investigate how to leverage the B-tree index on service operations to compute the service skyline in this section. We will use the service request in Example 2.1 to illustrate the idea. Since a SEP consists of two service operations (`drivingHistory`, `insuranceQuote`) and each operation has two QoWS parameters. Therefore, we have four attributes, i.e., `drivingHistory.fee`, `drivingHistory.latency`, `insuranceQuote.fee`, and `insuranceQuote.latency`. Assume that these four attributes are all indexed by using a B-tree. We start by scanning the four indices simultaneously to find the first match of the service id (i.e., *sid*). All the service instances that are not inspected before the first match will be removed from further computation. The remained service instances will go through a second round selection to determine the final skyline. The second round selection needs to be conducted in a brute-force manner. The quality of a SEP (i.e., fee and latency, etc) will be calculated using the aggregate functions defined in Table 1. Then a non-index based approach (e.g., block nested loop, divide-and-conquer, etc) can be applied to compute the final skyline.

Figure 4 illustrates how to use the B-tree approach to compute a skyline for the example service request. For example, the simultaneous scan identifies the first match, which is S_8. The SEPs that have not been inspected are then eliminated. The scan process guarantees that, for any SEP_k that has not been inspected, ($SEP_8 \rhd^{op_1} SEP_k$) $\wedge$ ($SEP_8 \rhd^{op_2} SEP_k$) is true, where op_1 and op_2 are `drivingHistory` and `insuranceQuote` respectively. According to Theorem 3.1, $SEP_8 \rhd^{(op_1,op_2)} SEP_k$ is true, i.e., SEP_8 dominates SEP_k. Therefore, SEP_k should be removed from further computation. The remaining SEPs (i.e., SEP_2, SEP_{15}, …,SEP_8) will go through the second round selection to generate the target skyline.

The above approach assumes that different SEPs have distinct values on each attribute so that they can be sorted in a strictly increasing order. When there are duplicate QoWS values, some SEPs may not be properly pruned in the first round and thus increase the computational overhead of the second round. This can be handled by using an approach, which is similar to the one proposed in [18]. In particular, a buffer B_i will be created for attribute A_i, where B_i is used to store

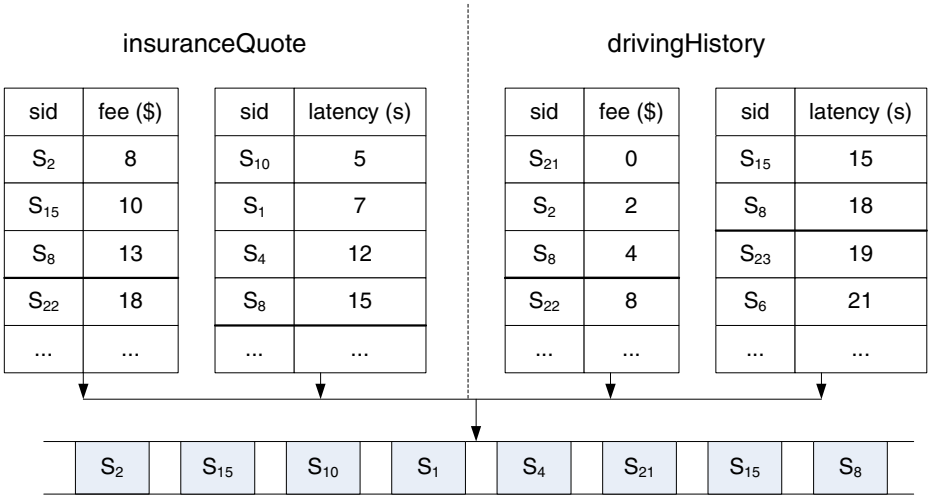

Figure 4 Indexing the service operations using a B-tree.

possible skyline SEPs that share the same value in A_i. Before inserting a potential skyline SEP into the buffer, it will first be compared with the SEPs within the buffer. If the new SEP is dominated, it won't be inserted. If any SEP in B_i is dominated, it will be removed from the buffer. When a SEP with a larger value in A_i is inserted into B_i, all SEPs within the buffer will enter the second round selection.

3.2 Indexing service operations using R-trees

In this section, we assume that the service operations are indexed by using a R-tree. We study how to extend the nearest neighbor algorithm [15] to compute service skylines in this section. We continue to use the service request in Example 2.1 to illustrate the computation process (see Figure 5). We divide the main procedure into five steps:

1. Take the first member operation (i.e., `drivingHistory`) and compute the nearest neighbor of data record $(0, 0)$. Assume that the nearest neighbor we get is S^1_{NN} and the set of SEPs dominated by S^1_{NN} is A. We then take S^1_{NN} and evaluate it on the second operation (i.e., `insuranceQuote`) and get the set of SEPs dominated by S^1_{NN}, which is B.
2. Compute the intersection of A and B and we get $C = A \cap B$. According to Theorem 3.1, all the SEPs in C are dominated by S^1_{NN} and therefore should be removed from further computation.
3. For SEPs that are not dominated by S^1_{NN}, recursively apply steps 1 and 2 to further remove dominated SEPs..
4. For the remaining SEPs, reverse the order of member operations (i.e., find the nearest neighbor of `insuranceQuote` and evaluate it on `drivingHistory`), and apply the above three steps.
5. Use a brute-force approach to generate the skyline from the remaining SEPs.

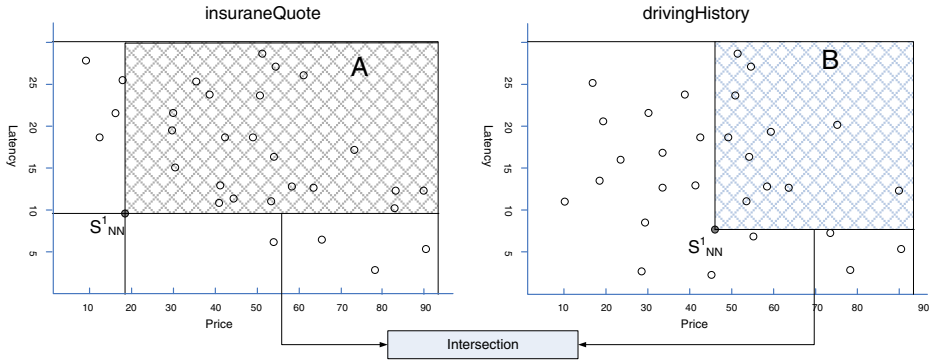

Figure 5 Indexing the service operations using a R-tree.

The above algorithm requires a frequent computation of intersections between two sets. Computing set intersections could be very expensive especially for large sets. Assume that the cardinalities of set A and B are k_1 and k_2 respectively. To get the intersection C, we usually need to do a pairwise comparison between these two sets that requires $O(k_1 \times k_2)$ time complexity. When k_1 and k_2 are large, this will be prohibitive as a frequently executed operation. Another approach is to sort the two sets and do a sequential pass simultaneously on them. This would require a time complexity of $O(k_1 \times \log k_1 + k_2 \times \log k_2)$.

We adopt the *key-indexed search* strategy to make this operation run in linear time. To achieve this efficiency, we need $O(n)$ additional space as a tradeoff. Suppose there are n service instances with the car insurance service. We assign the *id* numbers of these instances from 0 to $n-1$, i.e., $sid \in [0, n-1]$. The *id* numbers are distinct from each other. We use an assistance array T of size n to record these *sid*s. All the elements of the array are initialized to be 0. To compute the intersection of A and B, we first pass set A and add 1 the corresponding item of T accordingly. For example, if $sid_k \in A$, then set $T[sid_k]$ to 1 (the initial $T[sid_k]$ is 0). We then continue to apply the same process using B. In the end, all the *sid*s that satisfy $T[sid] > 1$ fall into the intersection. Obviously, the time complexity of this approach is $O(n)$.

We can further reduce the space complexity by using bitmaps to implement the assistance array T. Specifically, we can use two n-bits vectors V_1 and V_2 to replace T. They will only use 1/16 of the space used by T. All the bits in V_1 and V_2 are initialized to 0. Similarly, we first pass A and set the corresponding bits of V_1 to 1 accordingly, i.e., if $sid_k \in A$, then $V_1[sid_k] = 1$. When we pass B, we set the bits in V_2 according to V_1, i.e., if $(sid_k \in B) \wedge (V_1[sid_k] == 1)$, then $V_2[sid_k] = 1$. Finally, all the *sid*s that satisfy $V_2[sid] == 1$ fall into the intersection. We can use bitwise operators that enable to process bits in a batch mode to further improve the efficiency (Figure 6).

3.3 Analysis

Theorem 3.1 offers us heuristics for pruning dominated SEPs. However, algorithms based on this theorem may be far less efficient than an optimal algorithm because they have to use a brute-force approach on a usually large SEP space due to the

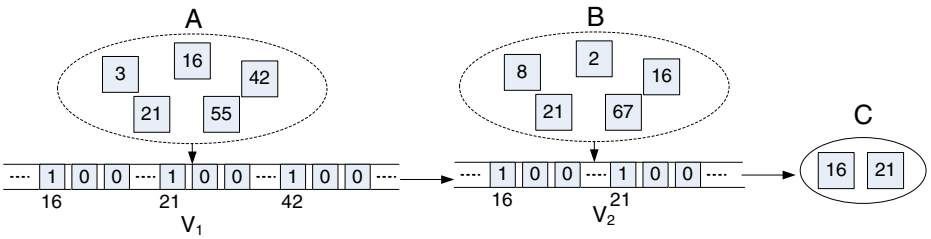

Figure 6 Computing set intersections using bitmap.

limited pruning power of the adopted heuristics. In this section, we conduct an analysis that gives an insight on some inherent issues.

We continue to use the service request in Example 2.1. Since each SEP consists of two operations, $SEP_1 \triangleright SEP_2$ actually means $SEP_1 \triangleright^{(op_1,op_2)} SEP_2$. There are four possible situations that lead to $SEP_1 \triangleright^{(op_1,op_2)} SEP_2$:

1. $(SEP_1 \triangleright^{op_1} SEP_2) \wedge (SEP_1 \triangleright^{op_2} SEP_2)$
2. $(SEP_1 \triangleright^{op_1} SEP_2) \wedge (SEP_1 \not\triangleright^{op_2} SEP_2)$
3. $(SEP_1 \not\triangleright^{op_1} SEP_2) \wedge (SEP_1 \triangleright^{op_2} SEP_2)$
4. $(SEP_1 \not\triangleright^{op_1} SEP_2) \wedge (SEP_1 \not\triangleright^{op_2} SEP_2)$

Theorem 3.1 only covers the first situation, i.e., $(SEP_1 \triangleright^{op_1} SEP_2) \wedge (SEP_1 \triangleright^{op_2} SEP_2) \Rightarrow SEP_1 \triangleright^{(op_1,op_2)} SEP_2$. We can directly eliminate SEP_2 if the first situation is satisfied. However, for the remaining three situations, we have to postpone the decision until the aggregate attributes of the SEPs are actually computed and compared. Therefore, the heuristics derived from Theorem 3.1 only help prune a (sometimes small) subset of SEPs. For a correlated service space, where some service instances that are good in one quality aspect are also good in the other quality aspects, the heuristics may have a good pruning capability because there may be few very good SEPs that dominate others. However, for other circumstances, the prune power may be very limited.

4 Indexing the SEPs

Based on the above analysis, we first present a *Baseline Algorithm* (BA) that relies on a two-level pruning mechanism to efficiently compute the service skyline. We then conduct an in-depth performance analysis, which helps reveal some inherent issues with this algorithm. This also leads us to develop a novel indexing structure on SEPs. The experimental study in Section 5 justifies the effectiveness of the proposed indexing structure.

4.1 The baseline algorithm

We propose in this section the BA algorithm to efficiently retrieve service skylines. We continue to use the service request in Example 2.1. Similar to the B-tree based approach, we treat the operation space as having four dimensions: $op_1.fee$, $op_2.latency$,

$op_1.fee$, and $op_2.latency$, where op_1 and op_2 represent `drivingHistory` and `insuranceQuote`, respectively. Suppose that this operation space is indexed by a R-tree, referred to as $\mathcal{R}^O$. The leaf nodes of the R-tree represent the actual SEPs. An intermediate node represents a minimum bounding rectangle (MBR) of each node at its lower level. Similar to BBS (branch and bound skyline) [20], BA also leverages a priority queue (or a heap) to make sure that the SEPs are enumerated in an ascending order of their *mindist*. The *mindist* of a leaf node is the summation of all its coordinate values whereas the *mindist* of an intermediate node is the *mindist* of its lower-left corner point. The heap is constructed to efficiently output the node (intermediate or leaf node) that has the minimum *mindist*.

Recall that if SEP_1 dominates SEP_2, it actually means $\text{SEP}_1 \rhd^{(op_1,op_2)} \text{SEP}_2$ and there are four possible situations that may lead to $\text{SEP}_1 \rhd^{(op_1,op_2)} \text{SEP}_2$ as stated in Section 3.3. Since $\mathcal{R}^O$ is an index that is built upon service operations, in essence, it can only deal with the first situation. Therefore, we need to make key extensions to BBS in order to compute the service skyline. The detailed algorithm is given in Algorithm 1. BA initially inserts all the entries in the root of $\mathcal{R}^O$ into the heap H. The entry with the minimum *mindist* is removed from the heap. This entry is then expanded and all its child entries are inserted into the heap. Again, the entry with the minimum *mindist* will be removed from the heap, expanded, and all its child entries will be inserted into the heap. This process continues until the first leaf node is removed by the heap. This leaf node will be inserted into the resultant skyline list L. A SEP R-tree, referred to as, $\mathcal{R}^S$, will then be initialized using the first skyline SEP. The purpose of $\mathcal{R}^S$ is to effectively prune dominated SEPs. More specifically, $\mathcal{R}^S$ will be dynamically constructed with two dimensions: SEP.fee and SEP.latency (because this is a R-tree on SEPs not operations), where

$$SEP.fee = op_1.fee + op_2.fee \tag{1}$$

$$SEP.latency = op_1.latency + op_2.latency \tag{2}$$

After $\mathcal{R}^S$ is constructed, the entries output from the heap will be checked against it for dominance. Specifically, if a top entry in the heap is dominated by some SEP in $\mathcal{R}^S$, it can be directly pruned. Otherwise, we have two situations:

1. If the entry is an intermediate node, it will be expanded into its child entries and these child entries will also be checked for dominance against $\mathcal{R}^S$ before inserting into the heap. The dominated entries can also be directly pruned.
2. If the entry is a leaf node. it will be inserted into both L and $\mathcal{R}^S$.

As can be seen in Algorithm 1, $\mathcal{R}^O$ and $\mathcal{R}^S$ essentially form a two-level pruning mechanism and the SEPs that cannot be pruned by these two R-trees are the skyline SEPs.

4.2 Analysis

$\mathcal{R}^S$ enables dominance checking on the aggregate QoWS attributes of SEPs. It determines the SEP *skyline search region (SSR)* that is the section of the data space

Algorithm 1 BA

Input: A R-tree RT
Output: A list of the SEP skyline points L

```
1:  L = φ, R^S = φ;
2:  insert all entries in the root node of R^O into H
3:  while H ≠ φ do
4:      e = H.extractmin();
5:      if R^S ≠ φ then
6:          map e to the dimensions of R^S and check dominance;
7:          if e is dominated then
8:              prune e;
9:          else
10:             if e is an intermediate node then
11:                 for each child entry e.c_i of e do
12:                     if e.c_i is not dominated by R^S then
13:                         H.insert(e.c_i);
14:                     end if
15:                 end for
16:             else
17:                 L.insert(e);
18:                 R^S.insert(e);
19:             end if
20:         end if
21:     else
22:         if e is an intermediate node then
23:             for each child entry e.c_i of e do
24:                 H.insert(e.c_i);
25:             end for
26:         else
27:             L.insert(e);
28:             initialize R^S using e;
29:         end if
30:     end if
31: end while
```

containing the skyline SEPs [20]. By observing formulae (1) and (2), we find that $\mathcal{R}^O$ that is used to index the operation space and $\mathcal{R}^S$ share the same *mindist*, i.e.,

$$mindist = SEP.fee + SEP.latency \tag{3}$$

$$= \underbrace{op_1.fee + op_2.fee}_{SEP.fee} + \underbrace{op_1.latency + op_2.latency}_{SEP.latency} \tag{4}$$

The *mindist* of a SEP is defined as the sum of its QoWS attributes. Different QoWS attributes are normalized into the same range (e.g., [0, 1]) before being aggregated. Based on (3) and (4), $\mathcal{R}^O$ and $\mathcal{R}^S$ share the same *mindist* as long as the aggregation functions are defined as the sum of the QoWS attributes from the SEP's member

operations. For some aggregation functions that are usually defined as the product of the QoWS attributes from the member operations (e.g., reliability and availability), we take a logarithm to make an adaption as shown in Table 1.[1]

Given that $\mathcal{R}^O$ and $\mathcal{R}^S$ share the same mindist function, BA only accesses *candidate* entries in $\mathcal{R}^O$ that potentially contain skyline SEPs. Here, we define a candidate entry as a R-tree entry that intersects with the *SSR*. A non-candidate entry, say e, does not overlap with the *SSR*, which also implies that there is a skyline SEP ψ that can dominate the lower-left corner of e. ψ must also have a *mindist* that is smaller than that of e [20]. We use the notion *mindist*($\mathcal{R}^S$) to denote the *mindist* calculated under $\mathcal{R}^S$. Similarly, we use the notion *mindist*($\mathcal{R}^O$) to denote the *mindist* calculated under $\mathcal{R}^O$. Recall that the heap enables that the entries in $\mathcal{R}^O$ are visited in ascending order of their *mindists*. If *mindist*($\mathcal{R}^S$) equals to *mindist*($\mathcal{R}^O$), it can be guaranteed that ψ is processed before e so that e is pruned by ψ. In this manner, the non-candidate entries will be pruned and only candidate ones are accessed.

Based on the above analysis, we can estimate the number of nodes accessed by BA. Assume that the height of $\mathcal{R}^O$ is h and there are $cand_i$ candidate nodes in the ith level of the R-tree. The total number of node accesses can be represented as

$$NA = \sum_{i=0}^{h-1} cand_i \tag{5}$$

To further examine how NA is related to the structure of the R-tree and the inherent characteristics of the data space, we further elaborate (5). Specifically, h can be specified as $1 + \lceil \log_f(\frac{N}{f}) \rceil$, where N is the cardinality of the data space and f is the average fanout of a node in $\mathcal{R}^O$. Suppose there are n_i nodes at level i and the probability that a node at level i intersects with SSR is $P^i_{\text{intsect(SSR)}}$. The candidate nodes at level i can be described as [20]

$$cand_i = n_i \times P^i_{\text{intsect(SSR)}} \tag{6}$$

The number of node at level i can be specified as $n_i = \frac{N}{f^{i+1}}$. $P^i_{\text{intsect(SSR)}}$ can be evaluated by using the node density $D_i(p)$ at level i, i.e.,

$$P^i_{\text{intsect(SSR)}} = \int_{p \in SSR} D_i(p)dp \tag{7}$$

Assuming that each leaf node visited contains some skyline SEPs, a pessimistic upper bound for retrieving the entire skyline is given by [20] , which is $|L| \times h$. It is decided by the cardinality of the skylines (i.e., $|L|$) and the height of $\mathcal{R}^O$. This upper bound corresponds to the situation that the algorithm needs to go through a complete path (i.e., the length of the path is h) to find each skyline point. However, multiple skyline points may be grouped into a single node or belong to the same branch of the R-tree. In this regard, the R-tree can be viewed as a *cluster* mechanism that groups together

[1] Some other monotonic aggregation functions can also be handled in a similar way, e.g., $\left(\prod_{i=1}^n x_i\right)^{1/k}$ and the exponential function $e^{\sum_{i=1}^n x_i}$. Since sum and product are the most commonly used aggregation functions for QoWS, we mainly focus on these two types of functions.

the points with similar properties (e.g., similar coordinate values). Since the total number of node accesses is less than $|L| \times h$, we can have

$$NA = \alpha \times |L| \times h = \sum_{i=0}^{\lceil \log_f(\frac{N}{f}) \rceil} \frac{N}{f^{i+1}} \times \int_{p \in SSR} D_i(p)dp \tag{8}$$

where $\alpha \in (0, 1]$ is defined as a bounding factor. From (5), (6) and (7), we can see that NA is determined by the cardinality of the data space, the node density of each level of the R-tree, and the fanout of the R-tree. Therefore, α is related to the structure of the R-tree and the inherent characteristics of the data space.

The above analysis helps us further investigate the performance of BA. Although it has an upper bound of $|L| \times h$, it may not be an optimal solution for retrieving the SEP skylines (i.e., the bounding factor α of BA may be large). The reason is that BA is based on a R-tree (i.e., $\mathcal{R}^O$) that is constructed from the operations space. The operation space is different from the SEP space whose coordinates are the aggregates of the operations. Therefore, (1) the SEP space may have different characteristics with the operation space; (2) a R-tree built from the SEP space (where we try to find the skylines) may have a different structure with a R-tree built from the operation space. We illustrate these two aspects by using two simple examples.

4.2.1 Space characteristics

Based on formula (2), the latency of a SEP is the sum of the latency from its member operations. We use X and Y to represent the latency of the two member operations and assume they are two independent continuous random variables with density functions $f_X(x)$ and $f_Y(y)$. The latency of the SEP is described as the sum of X and Y, i.e., $Z = X + Y$. The density function of Z is $f_Z(z)$ with $f_Z = f_X * f_Y$, where $*$ is the *convolution* operator. Specifically,

$$f_Z(z) = (f_X * f_Y)(z) = \int_{-\infty}^{+\infty} f_X(z - y) f_Y(y)dy$$

As an example to show how the distribution of Z is different from X and Y, we assume X and Y are randomly chosen variables from interval [0,1] with uniform probability density. $Z = X + Y$ is the sum of these two variables. The density functions of X and Y are described as:

$$f_X(x) = f_Y(y) = \begin{cases} 1 & \text{if } 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

The density function of Z can be computed from the *convolution* of f_X and f_Y:

$$f_Z(z) = \int_{-\infty}^{+\infty} f_X(z - y) f_Y(y)dy = \begin{cases} z & \text{if } 0 \le z \le 1 \\ 2 - z & \text{if } 1 \le z \le 2 \\ 0 & \text{otherwise} \end{cases}$$

We can see that Z has a triangle distribution that is different from both f_X and f_Y. We use this example just to show that how the characteristics of the SEP space may be different from the operation space.

4.2.2 R-tree structure

Using Example 2.1, we assume that all the service instances provide the insurance quote for free (i.e., $op_2.fee = 0$). This enables us to visualize the SEPs in the operation space (with three dimensions because the op_2.fee dimension is collapsed into the origin). As we can see in Figure 7, the operation space is represented using three coordinates: fee, $op_1.latency$, and $op_2.latency$. We select three representative SEPs, a, b, and c to investigate how they could be organized differently in the operation space and the SEP space. In the operation space, a and b are far from each other because they have quite different values for the latency on their two member operations, i.e., a is much more efficient on performing op_2 whereas b is much more efficient on performing op_1. Another SEP c is much nearer to a than b although it is less efficient on performing both op_1 and op_2 than a. Therefore, a and c may be more likely to be "clustered" into the same MBR by a R-tree built from the operation space. However, in the SEP space (represented using two coordinates: fee and $op_1.latency + op_2.latency$ in Figure 7), a' is actually much closer to b' than c'. In this case, a' and b' may be more likely to be "clustered" into the same MBR by a R-tree built from the SEP space. If both a' and b' belong to the service skyline, they can be retrieved together. In contrast, retrieving a and b from the operation space may require more node accesses because they could be in different leaf nodes or even different branches of the R-tree.

4.3 Operation graph based indexing (OGI)

An effective improvement on BA is to make it perform on a R-tree that is constructed directly from the SEP space. However, the challenge is that the SEP space is dynamically generated by each service request. This makes the SEP space inherently different from the operation space which is relatively static. Pre-computing an index structure for such a dynamic space seems to be infeasible.

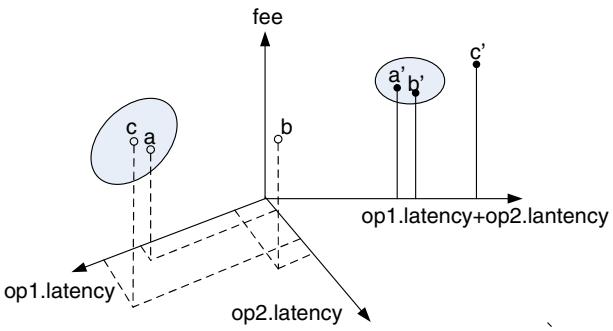

Figure 7 SEP distribution in operation space and SEP space.

In this section, we present an operation graph based indexing (OGI) approach to build indices for SEPs. Although different SEP spaces can be dynamically formed for different service requests, SEPs are not generated in an *ad hoc* manner. The dependency constraints between service operations only allow SEPs that conform to certain *patterns* to be generated. These patterns are like rules that define what kind of SEPs can be generated. If we know these patterns in advance, we can *foresee* the properties (i.e., the aggregate attributes) of the SEPs and construct index structures on them. Now the problem turns out to be whether such patterns exist or not and if yes, how to find such patterns. The proposed service model provides a natural solution for this problem.

Consider a set of SEPs, $\text{SEP}_1, \ldots, \text{SEP}_k$, that are generated from service relation SR with k service instances. Assume that these SEPs are used to access operation op specified by some service request. The operation graph $G(op)$ is made up of a minimum number of necessary operations that make op accessible. In another word, $G(op)$ consists of all operations that op depends on (i.e., ancestors of op in the service graph SG) and no other redundant operations. We can formally prove it as follows. Assume there exists an operation op' where op' is ancestor node of op in SG and op' is not a node of $G(op)$. Let P_1 be the path from op' to op. Also, we denote the path from ϵ to op' as P_2. Connecting P_1 and P_2, we get a new path P_3 from ϵ to op, which is not included in $G(op)$. This contradicts definition of the operation graph. Therefore, no such op' exists. Thus, $G(op)$ includes all operations that op depends on. Let's now assume that there exists a graph $G'(op) = G(op) - \{op'\}$, where $op' \neq op$, such that $G'(op)$ includes all operations that op depends on. From definition of the service graph, there is a path to op which passes through op' in SG. Therefore, op' is an ancestor node of op in SG. Since $G'(op)$ includes all operations that op depends on, we have $op' \in G'(op)$. This contradicts the fact that op' is removed from $G'(op)$. Therefore, $G(op)$ does not include any redundant operations.

The above proof shows that $G(op)$ captures all the operations (i.e., op and all its the operations it depends on) in the SEPs. On the other hand, the service instances in SR store the QoWS parameters for each of these operations. Thus, the operation graph $G(op)$ and service relation SR carry enough information to construct the index for the SEPs. The algorithm for constructing OGI is presented in Algorithm 2.

Algorithm 2 OGI_construction

Input: A requested operation op, a service graph $SG(V, E)$, and a service relation SR
Output: A R-tree RT for the SEP space

1: $OG = G(op)\_construction(op, SG)$;
2: $\mathcal{S} = \phi$; //a two dimensional array used to store the SEPs
3: **for** each $op \in V[OG]$ **do**
4: **op** = select op from SR; //retrieve the QoWS of op from all service instances in SR
5: insert **op** into $\mathcal{S}$ as a column;
6: **end for**
7: $RT = build\_Rtree(\mathcal{S})$; //build the R-tree from S, where each row of $\mathcal{S}$ stores the QoWS for a SEP

Algorithm 3 $G(op)$_construction

Input: A requested operation op and a service graph $SG(V, E)$
Output: An operation graph $OG(V, E)$

```
1:  V[OG] = φ; E[OG] = φ; //initialize the operation graph OG
2:  O = {op}; //O is a set for storing operations
3:  while O ≠ φ do
4:     op = O.remove(); //remove an operation op from O
5:     V[OG] = V[OG] ∪ {op};
6:     for each (o, o') ∈ E[SG] do
7:        if (o' == op) then
8:           E[OG] = E[OG] ∪ {(o, o')};
9:           O = O ∪ {o};
10:       end if
11:    end for
12: end while
```

Operation graph based indexing (OGI) enables us to directly construct indices for SEPs when the service schema is available. Algorithms built from OGI can thus use an index on the SEP space. This has two major advantages over BA:

- OGI overcomes the "distortions" introduced by using an index on the operation space.
- A R-tree index on the SEP space has lower dimensionality than a R-tree on the operation space. The former has a dimensionality that equals to the number of user interested quality attributes while the dimensionality of the latter equals the number of user interested quality attributes times the service operations in a SEP.

4.4 Incorporating space partition tree

As the size of the service skyline increases, a large number of SEPs (including all skyline SEPs) may not be pruned by the indices. Instead, pairwise comparisons have to be performed to determine the skyline membership. Even if a main-memory R-tree can be used, the R-tree still needs to be frequently updated due to the insertion of each skyline SEP, which can also be computationally expensive.

Inspired by the recent works on skyline computation in the database community [17, 36], we propose to consider *incomparability* among the SEPs when computing the final service skyline. The R-tree that is built upon the operation space (for BA) or the SEP space (for OGI) provides an effective way to perform dominance checking. It groups together SEPs with similar QoWS so that they can be efficiently pruned in a batch mode. On the other hand, for the SEPs that cannot be pruned, as most of them are skyline SEPs, they are not comparable to each other (or they do not dominate each other). As the size of the skyline increases, it demands a large number of pairwise comparisons, which are computationally expensive and slow down the entire skyline computation process. We propose a hybrid structure that combines a R-tree with a space partition tree. The space partition tree is similar to the one proposed in [36], which is mainly used to efficiently process SEPs that are not pruned

by the R-tree. The hybrid structure thus optimizes both dominance checking (using the R-tree) and incomparability checking (using the partition tree) and is expected to achieve good performance and scalability. In what follows, we briefly describe the space partition tree and then present how the R-tree and the space partition tree can be integrated into the skyline computation process.

Given d user interested QoWS parameters, the SEP space will be divided into 2^d partitions by a given skyline SEP, called a reference point. The key idea is that no pairwise comparison is needed between SEPs that are assigned to incomparable partitions. For example, as illustrated in Figure 8a, the SEP space is divided by skyline SEP S_0 into four partitions, which are addressed as 00, 01, 10, and 10, respectively. No pairwise comparisons are needed between SEPs in partition 10 (i.e., S_1 to S_4) and the ones in partition 01 (i.e., S_5 to S_8) because they are guaranteed to be incomparable to each other. Following the same lines, the SEP space can be recursively divided until no further partitioning can be performed. For example, S_2 and S_6 can be used to further divide partitions 01 and 10. As the number of incomparable SEPs increases, pairwise comparisons can be significantly reduced, which will greatly improve the performance. The partitions can be efficiently represented by the space partition tree, as shown in Figure 8b. Each node has no more than one child and the child node represents the reference point from the partition with the smallest address. For example, S_6 is the child node of S_0 because S_6 has a smaller partition address than S_2. The reference points from other partitions form the sibling nodes, which are ordered in ascending partition address.

The space partition tree will be initialized when the first skyline SEP is identified. The entries output from the heap will then be evaluated using the partition tree. If the entry is dominated (i.e., fall into the dominated partition such as 11 in Figure 8a), it will be pruned. Otherwise,

1. If the entry is an intermediate node, it will be expanded into its child entries and these child entries will also be evaluated using the partition tree before inserting into the heap. The dominated entries can also be directly pruned.
2. If the entry is a leaf node. it will be inserted into the partition tree.

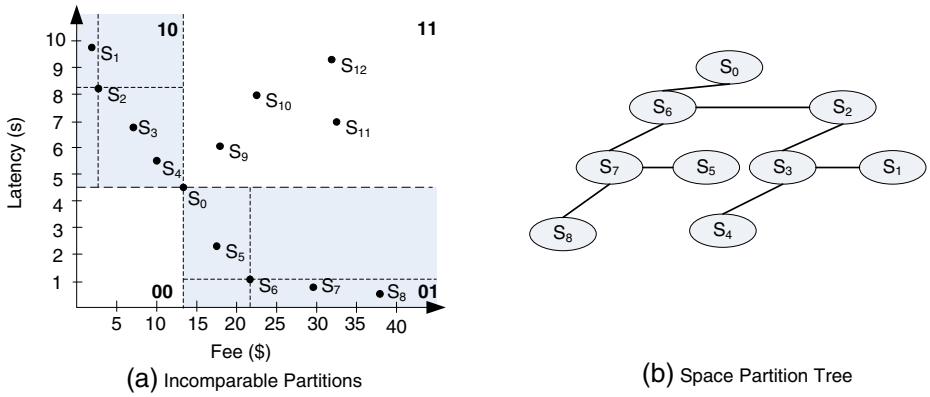

Figure 8 Incomparable partitions in the SEP space.

4.5 Computing SEP skylines over multiple services

The approaches we presented so far focus on computing service skylines from a single service. In Example 1.1, Mary wants to access an integrated service package that consists of three services: *Car Insurance, Car Purchase,* and *Financing*. Processing such a service request will generate SEPs that are across multiple services. We define a service skyline that is computed from multiple service relations, $\mathrm{SR}_1, \ldots, \mathrm{SR}_m$, as a multi-service skyline, denoted as SK_{all}. Suppose that there are $k_1, \ldots, k_m$ service tuples for each of the m service relations. To find the skyline for the service package, $N = \prod_{i=1}^{m} k_i$ number of candidate SEPs need to be evaluated. The computational cost would be prohibitive if the number of service instances in each service relation is large. Fortunately, we can leverage the following key property of service skylines to greatly improve the performance.

Lemma 4.1 *Consider a set of service relations $SR_1, \ldots, SR_m$, and a set of service skylines $SK_1, \ldots, SK_m$, computed for each of them. A multi-service skyline SK_{all} over $SR_1, \ldots, SR_m$ can be completely decided by $SK_1, \ldots, SK_m$.*

Proof sketch Lemma 4.1 essentially says that any multi-service SEP, $\psi \in \mathrm{SK}_{all}$, must be aggregated from a set of SEPs, $\psi_1, \ldots, \psi_m$, where each ψ_j is a skyline SEP from SR_j. Assume that ψ_j is not a skyline SEP from SR_j. Thus, we can always find a skyline SEP, say ψ'_j, from SR_j, such that ψ'_j dominates ψ_j. Therefore, we can get a multi-service SEP, say ψ', by aggregating ψ'_j with $\psi_1, \ldots, \psi_{j-1}, \psi_{j+1}, \ldots, \psi_m$. Thus, ψ' dominates ψ. This contradicts the fact that ψ is in the skyline SK_{all}. □

Lemma 4.1. enables us to compute the skylines over multiple services by using the skylines from each individual services. Since the size of a skyline is expected to be much smaller than the number of service instances, a large portion of computation overhead can be effectively reduced. Computing a multi-service skyline over m services can be conducted in two conceptually separated steps:

1. Compute the service skylines for each individual service. The cost of this step is dominated by the total number of node accesses, which can be quantified as $\sum_i^m NA_i$.
2. Pick one SEP from each skyline resulted from the first step to generate the multi-service skyline and enumerate all possible combinations. The cost of this step is determined by $\prod_i^m |L_i|$, where $|L_i|$ is the size of the skyline for the ith service.

5 Experimental study

We conduct an extensive set of experiments to assess the effectiveness of the proposed service skyline computation algorithms. We run our experiments on a Macbook Pro with 2.5 GHz Intel Core 2 Duo processor and 4G Ram under Mac OS X 10.5.8. Since there is not any sizable Web service test case that is in the public domain and that can be used for experimentation purposes, we focus on evaluating the proposed skyline algorithms and indexing structure by using synthetic

Table 2 Abbreviation of algorithms.

BA-IR	Baseline algorithm using an main memory R-tree
BA-PT	Baseline algorithm using a space partition tree
OGI-IR	OGI algorithm using an main memory R-tree
OGI-PT	OGI algorithm using a space partition tree
LESS	Linear Elimination Sort for Skyline [11]
OSP	Object-based space partitioning algorithm [36]

Web services. The QoWS attributes[2] of syntactic service instances are generated in three different ways following the approach described in [5]: (1) *Independent QoWS* where all the QoWS attributes of service instances are uniformly distributed, (2) *Anti-correlated QoWS* where a service instance is good at one of the QoWS attributes but bad in one or all of the other QoWS attributes, and (3) *Correlated QoWS* where a service instance which is good at one of the QoWS attributes is also good at the other QoWS attributes.

We setup a set of experiment parameters to evaluate and compare the performance of BA and OGI. These include the number of QoWS attributes (i.e., d) in the range of 2–10, the number of operations per SEP (i.e., o) in the range of 2–10, and the cardinality of the service relations (i.e., n) in the range of 100k–500k (i.e., 100,000 to 500,000). We also study the performance of skyline computation over multiple services and investigate how the performance varies with different number of services in a SEP. By performance, we report both the node accesses (which is independent of hardware settings) and the actual running time on our experiment machine. Finally, we study the sizes of the SEP skylines and examine whether they are in a practical range for user selection. To further evaluate the performance of the proposed algorithms, we also implemented and compared our algorithms with LESS [11] and the object-based space partitioning algorithm (OSP) [36]. For LESS, we use an EF window that holds 100 SEPs in the first pass. For OSP, we implemented the sorting first version of the algorithm. All algorithms in Table 2 are implemented in Java.

5.1 Effect of the space partition tree

We first investigate the effectiveness of the space partition tree. We compare it with the main memory R-tree based approach described in Section 4.1. Some interesting observations are summarized as follows: (1) For anti-correlated QoWS with high dimensionality (Figure 9a), the space partition tree helps generate the skyline in a much more efficient manner than the main memory R-tree: OGI-PT and BA-PT outperform OGI-IR and BA-IR, respectively for $d \geq 6$). (2) For independent QoWS, the advantage of using a space partition tree is not as obvious as the anti-correlated case: OGI-PT is slightly more efficient than OGI-IR for $d \geq 9$. (3) For correlated QoWS, using a main memory R-tree has a better performance than the space partition tree.

The results can be interpreted as follows. For the anti-correlated case, as the dimensionality increases, the size of the skyline will quickly increase (also see

[2]We use QoWS attributes instead of QoWS parameters in the experiment section to differentiate it from the term "experiment parameters" we use in this section.

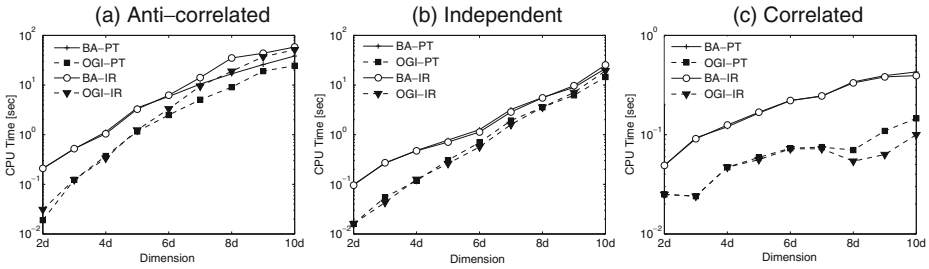

Figure 9 Effect of partition tree ($n = 100$k, $o = 2$).

Figure 17). Hence, the number of incomparable SEPs will increase accordingly. As described in Section 4.4, the space partition tree can efficiently process incomparable SEPs and thus achieve a much better performance than a memory-based R-tree. For the case of correlated QoWS, as the size of the skyline is small, the number of incomparable SEPs is limited. The memory-based R-tree can group together similar SEPs and prune them in a batch mode, which makes it more efficient than the space partition tree. Nevertheless, as shown in Figure 9c, both the space partition tree and the main-memory R-tree can generate the skyline in a rather efficient manner. The above observations justify that the hybrid structure that combine a R-tree with a space partition tree provides a good balance between dominance checking and incomparability checking and thus can achieve a good performance for different data distributions.

5.2 Number of QoWS attributes

We study the effect of the number of QoWS attributes in this section. We keep the cardinality as 100k, the number of operations per SEP as 2, and vary the number of attributes from 2 to 10. Figures 10 and 11 show how the number of node accesses and the actual running time vary with the number of attributes for the three different QoWS distributions. For both anti-correlated and independent QoWS, OGI-PT outperforms BA-PT on low dimensionality by almost an order of magnitude but the difference decreases as the number of attributes increases. The performance difference comes from two sources: (1) BA-PT operates on a R-tree built from the operation space as contrast to a R-tree built from the SEP space used by OGI-PT;

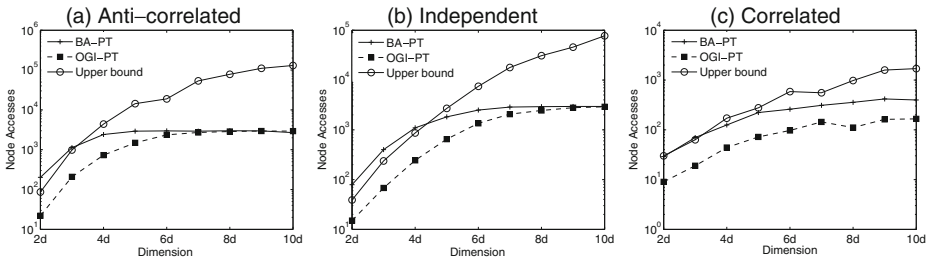

Figure 10 Node accesses vs. number of attributes.

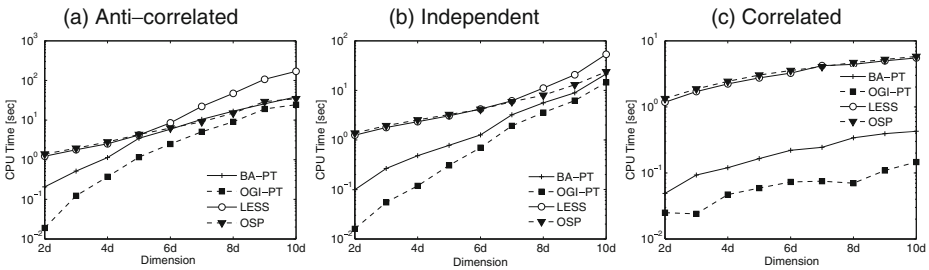

Figure 11 CPU time vs. number of attributes.

(2) The R-tree used by BA-PT has a dimensionality which is two times (since the number of operations per SEP is 2 in this case) of the one used by OGI-PT. The difference becomes smaller with a larger number of attributes, which is because both algorithms are dominated by the poor performance of R-tree in high dimensions.

We also compare the number of node accesses with the theoretical upper bound described in Section 4.2. As can be seen in Figure 10, the number of node accesses is much lower than the upper bound except for some low dimensional cases. The reason is that the upper bound holds when we assume that each leaf node contains some skyline SEPs. However, when the dimensionality is low, the number of skyline SEPs is small and a lot of leaf nodes do not contain any skyline SEP.

We study the performance of BA-PT and OGI-PT by comparing them with LESS and OSP in Figure 11. OGI-PT achieves the best performance in all cases. BA-PT also outperforms LESS in all cases. For both anti-correlated and independent QoWS, the performance of BA-PT, OGI-PT, and OSP tend to be similar with the increase of dimensionality. This is because for these two distributions, the size of the skyline grows quickly with dimensionality. As more SEPs become incomparable to each other, the number of SEPs that can be pruned by the R-tree will be significantly reduced. In this case, both BA-PT and OGI-PT will rely on the space partition tree to process most SEPs, which explains their similar performance with OSP. For correlated QoWS, the performance of OGI-PT and BA-PT significantly outperforms LESS and OSP because a large number of SEPs can be directly pruned by the R-tree through dominance checking.

5.3 Number of operations per SEP

We study the effect of the number of operations per SEP with Figures 12 and 13. We keep the cardinality as 100k, the number of QoWS attributes as 2, and vary the number of operations per SEP from 2 to 10. OGI-PT is more efficient than BA-PT with several orders of magnitude and the difference increases with the number of operations. The performance degradation of BA-PT is mainly due to the dimensionality increment of the R-tree with the number of operations. A close investigation reveals that the performance of OGI is insensitive to the number of operations. By using the operation graph to index the SEPs, the same QoWS attributes from multiple operations (e.g., the fee of operations) are aggregated into a single QoWS attribute of the SEP (e.g., the fee of a SEP). Therefore, the

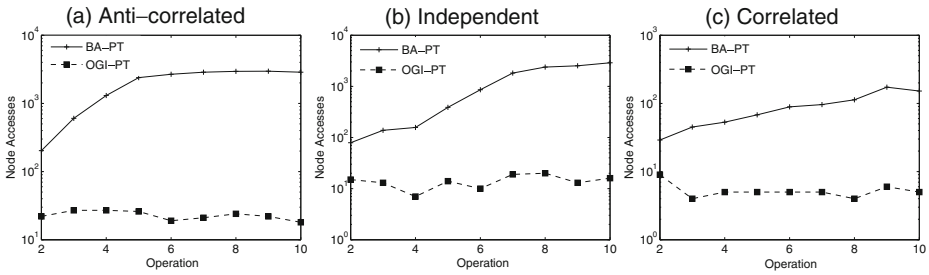

Figure 12 Node accesses vs. number of operations.

dimensionality of the R-tree used by OGI-PT equals to a constant (i.e., the number of different QoWS attributes which is 2 in this case) and will not increase with the number of operations.

For CPU time, OGI-PT achieves the best performance in all cases. For anti-correlated and independent QoWS with large number of operations, BA-PT uses similar or even more CPU time as compared with LESS and OSP for a large number of operations. The reason is similar to the one described above. When the number of operations is large, BA-PT uses a R-tree with a very high dimensionality. Thus, the number of node access increases accordingly, which affects the overall performance of BA-PT.

5.4 Cardinality of service relations

We show the effect of cardinality in Figures 14 and 15. We keep the number of QoWS attributes as 2, the number of operations per SEP as 2, and vary the cardinality from 100k to 500k. The performance of OGI-PT is consistently better than BA-PT due to similar reasons as described in Section 5.2. BA-PT also significantly outperforms LESS and OSP for independent and correlated QoWS. The number of node accesses of both BA-PT and OGI-PT and the CPU time of all algorithms tend to increase with the cardinality. For some cases, BA-PT and OGI-PT even perform more efficiently with larger cardinality. This may be caused by the positions of the skyline SEPs and the order they are retrieved [20].

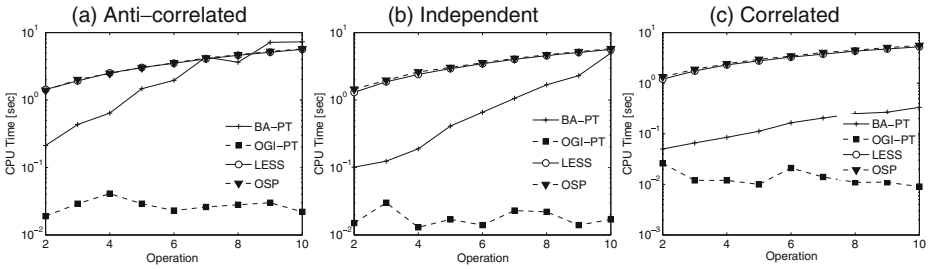

Figure 13 CPU time vs. number of operations.

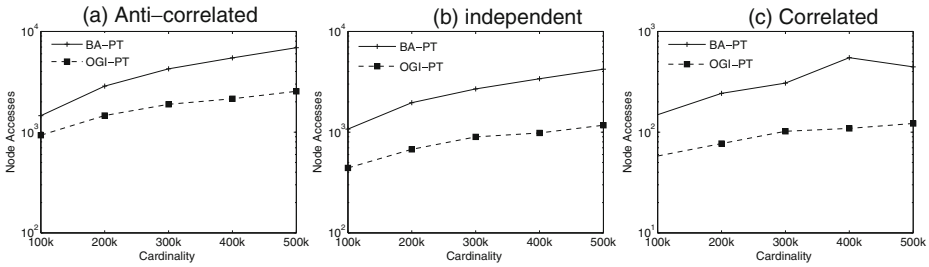

Figure 14 Node accesses vs. cardinality.

5.5 SEP skylines over multiple services

We now investigate the performance of skyline computation over multiple services. We keep the cardinality of each service relation as 100k, the number of QoWS attributes as 2, the number of operations per sub-SEP from each service as 2, and vary the number of services from 2 to 5. We follow the two-step procedure described in Section 4.5 and adopt OGI-PT in the first step to retrieve the skylines from each individual service. Figure 16 shows both the number of node accesses and the running time versus the number of services. The number of node accesses increases in a linear manner because the number of node accesses for m services is the sum of those for each individual service. The running time, however, increases in an exponential manner because the cost of the second step is determined by the product of the sizes of individual skylines. The overhead will be very obvious when the sizes of individual skylines are large. This applies for the anti-correlated QoWS which is expected to have a relatively large number of skyline SEPs. However, for the most practical scenarios where the number of services is no greater than three, the SEP skylines can still be generated very efficiently.

5.6 Sizes of the SEP skylines

We finally examine how the sizes of SEP skylines change with cardinality, number of operations per SEP, number of QoWS attributes, and number of services. Figure 17 presents some interesting effects of these parameters on the sizes of SEP skylines.

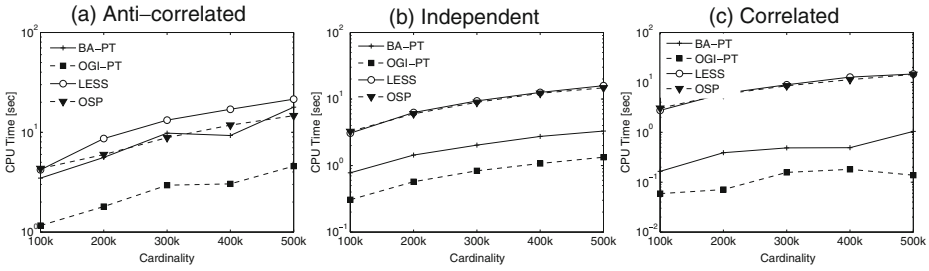

Figure 15 CPU time vs. cardinality.

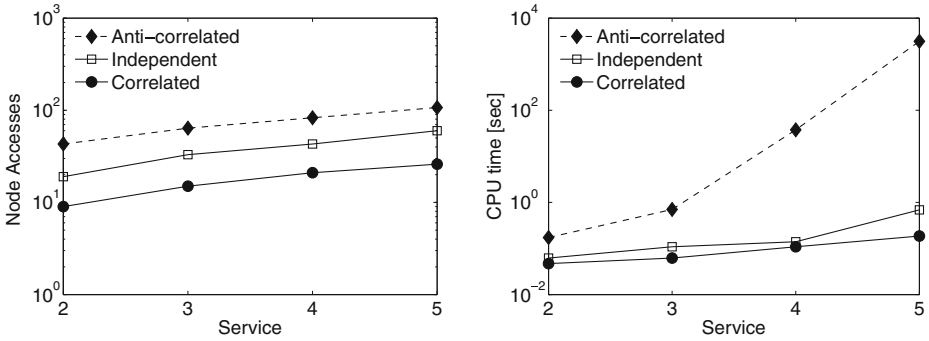

Figure 16 Performance vs. number of services.

First of all, the skylines generated from anti-correlated QoWS have larger sizes than those generated from independent and correlated QoWS, which is just as expected. Second, cardinality and number of operations per SEP have no obvious effect on the sizes of SEP skylines. As the cardinality varies from 100k to 500k, the sizes of skylines for independent and correlated QoWS vary from 5 to 20 whereas those for anti-correlated QoWS vary from 30 to 40. The sizes of skylines stay in a similar

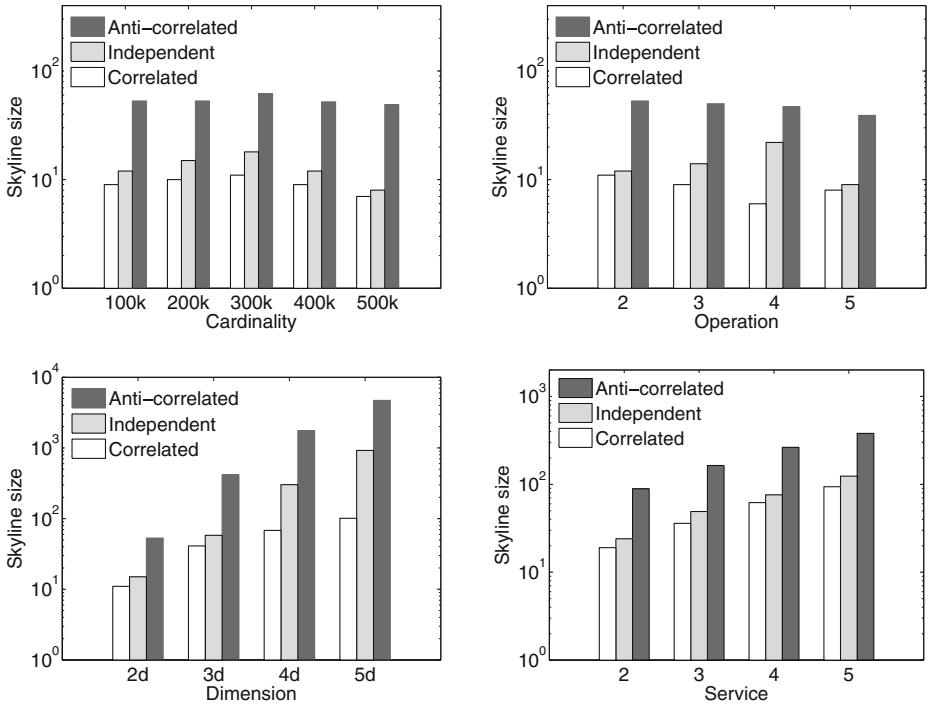

Figure 17 Sizes of SEP skylines.

range respectively when we vary the number of operations per SEP from 2 to 5. Interestingly, the sizes of skylines for anti-correlated QoWS have a trend to decrease with the number of operations. This may be due to that the aggregation of QoWS attributes from multiple operations compromises the anti-correlated effect. Third, the sizes of skylines clearly increase with the number of QoWS attributes and the number of services. However, in most practical usage scenarios where the number of QoWS attributes and the number of services are less than three, the sizes of the skylines are still within a practical range for user selection.

6 Related work

The proliferation of Web services is fostering a very active research area. We give an overview of the most closely related work.

In [26], a Web Service Management System (WSMS) is proposed to enable optimized querying over Web services. It incorporates Web services into the traditional Select-Project-Join queries and treats them as a type of expensive predicates [13]. An algorithm is proposed to arrange service calls into a pipelined execution plan. The optimization is "performance centered" that focuses on reducing the total running time. In [19], a query model is proposed that offers query optimization functionalities for Web services. The query model consists of three levels: *query level, virtual level*, and *concrete level*. The query model uses the predefined mapping rules to map relations defined at the query level to virtual operations defined at the virtual level. Users can thus directly use relations to query Web services. In [35], a composite service optimization approach is proposed based on several quality of service parameters. Composite services are represented as a state-chart. The optimization problem is tackled by finding the best Web services to execute a composite service in the form of a linear programming problem. In addition, the optimization approach adopted by [35] is based on the computation of a single objective function, which requires users to assign weights over different quality parameters. This also limits its applicability.

Skyline analysis, which was originally investigated as a mathematical problem [16, 22], was first introduced into the database domain by [5]. Three basic algorithms including block nested loops (BNL), divide-and-conquer, and B-tree based approach were presented to tackle the skyline computation problem [5]. The BNL algorithm was further extended with a pre-sorting scheme to improve the efficiency [8, 11]. A special function is adopted in [2] that sorts the data points based on their minimum coordinate value, which avoids the scanning of the entire dataset. Data incomparability has been considered as a key factor by some recent works in database skyline computation [17, 36]. An object-based space partition scheme is proposed in [36] that divides the data space into different partitions based on a reference skyline point. Data points that fall into incomparable partitions do not need to be compared with each other. A left-child/right sibling skyline tree is further developed that provides high space efficiency and fast access to the partitions and skyline points. To select the optimal skyline point to partition the data space, a cost model is developed in [17]. The selected pivot point is able to effectively prune non-skyline points by dominance and bypassing unnecessary dominance tests on incomparable pairs of points. Indexing structures have also been leveraged to improve the performance of skyline analysis. Two indexing structures were presented

in [27] with the ability to progressively report the skyline. NN and BBS are another two representative algorithms that can progressively process the skyline based on a R-tree indexing structure [15, 20]. Skyline computation has also been extended to a distributed environment, where data points are stored in different Web accessible databases [1]. A progressive distributed skyline algorithm was proposed in [18] that can progressively report the skyline points in a distributed environment.

Skyline or similar concepts have been applied in the area of service computing. A service discovery framework was developed in [24] that integrates the similarity matching scores of multiple service operation parameters obtained from various matchmaking algorithms. The framework relies on the service dominance relationships to determine the relevance between services and users' requests. Instead of using a weighting mechanism, the dominance relationship adopts a skyline-like strategy that simultaneously considers the matching scores of all the parameters for ranking the relevant services. A concept, called p-dominant skyline, was proposed in [32] that integrates the inherent uncertainty of QoWS in the service selection process. A p-R-tree indexing structure and a dual-pruning scheme were also developed to efficiently compute the p-dominant skyline.

Another possible solution to tackle service selection is to use top-k queries which have also received considerable attention recently. The top-k queries retrieve the best k objects instead of returning a single optimal object. This greatly reduces the decision space and also gives users certain flexibility to make their own choice among the k objects. Typical techniques for solving top-k queries include PREFER [14] and Onion [7] that rely respectively on pre-materialization and convex hulls. However, top-k queries are usually based on some specific preference function. Therefore, using top-k queries is not able to completely free users from assigning weights to different QoWS parameters.

7 Conclusion and future work

We present a novel concept, called service skyline, to tackle the service selection problem. The service skyline offers two significant benefits over existing service selection approaches. First, it completely frees service users from the weight assignment task in service selection. Second, it won't lose any merit of computing an objective function. A service skyline guarantees to include all user desired service providers. Since the service operations are relatively static, we first investigate how to leverage the indices on service operations to compute service skylines. The results led us to develop a Baseline Algorithm and a novel indexing structure for the dynamic SEP spaces. Analytical and experimental results show that the proposed indexing scheme is quite effective and efficient. For future work, we identify several important and promising directions:

- OGI is based on a R-tree index, which is optimized for a fixed set of dimensions (and operations in the context of service skyline). Their performances will decrease for skyline queries targeting different attributes (or operations). This limitation also applies to other index-based skyline approaches [28]. A straightforward extension that builds an index on all dimensions (or operations) suffers the issue of "curse of dimensionality" [3]. A suitable solution for the service skyline problem is to identify the typical usage patterns of service users. This

is practical for specific service domains because the user interested operations and QoWS attributes usually converge to a small number of candidates. A more general solution may be to extend the Skyline Cube approach [21, 34] and adapt it to the service skyline problem.

- Our aggregation functions do not take into consideration of missing quality values that may be common in real-world scenarios. Work on fuzzy-set based querying (e.g., SQL-F [6]) may be relevant for handling the situation of missing values.
- As the number of services increases, the running time of computing a service skyline over multiple services increases in an exponential manner. Another interesting future direction is to develop algorithms that can efficiently compute the service skyline over a large number of services.

References

1. Balke, W.-T., Guntzer, U., Zheng, J.X.: Efficient distributed skylining for web information systems. In: EDBT, pp. 256–273 (2004)
2. Bartolini, I., Ciaccia, P., Patella, M.: Efficient sort-based skyline evaluation. ACM Trans. Database Syst. **33**(4), 1–49 (2008)
3. Berchtold, S., Keim, D.A., Kriegel, H.-P.: The X-tree: an index structure for high-dimensional data. In: VLDB (1996)
4. Bianchini, D., De Antonellis, V., Melchiori, M.: Flexible semantic-based service matchmaking and discovery. World Wide Web **11**(2), 227–251 (2008)
5. Borzsonyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE (2001)
6. Bosc, P., Pivert, O.: SQLf: a relational database language for fuzzy querying. IEEE Trans. Fuzzy Syst. **3**(1), 1–17 (1995)
7. Chang, Y.-C., Bergman, L., Castelli, V., Li, C.-S., Lo, M.-L., Smith, J.R.: The onion technique: indexing for linear optimization queries. In: SIGMOD (2000)
8. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: ICDE (2003)
9. Cohen, S., Nutt, W., Serebrenik, A.: Rewriting aggregate queries using views. In: PODS (1999)
10. Dong, X., Halevy, A.Y., Madhavan, J., Nemes, E., Zhang, J.: Simlarity search for Web services. In: VLDB Conference (2004)
11. Godfrey, P., Shipley, R., Gryz, J.: Maximal vector computation in large data sets. In: VLDB (2005)
12. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In: VLDB (1995)
13. Hellerstein, J.M., Stonebraker, M.: Predicate migration: optimizing queries with expensive predicates. In: SIGMOD, pp. 267–276. ACM, New York (1993)
14. Hristidis, V., Koudas, N., Papakonstantinou, Y.: Prefer: a system for the efficient execution of multi-parametric ranked queries. In: SIGMOD (2001)
15. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: an online algorithm for skyline queries. In: VLDB (2002)
16. Kung, H.T., Luccio, F., Preparata, F.P.: On finding the maxima of a set of vectors. J. ACM **22**(4), 469–476 (1975)
17. Lee, J., Hwang, S.-W.: Bskytree: scalable skyline computation using a balanced pivot selection. In: EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology, pp. 195–206. ACM, New York (2010)
18. Lo, E., Yip, K.Y., Lin, K.-I., Cheung, D.W.: Progressive skylining over web-accessible databases. Data Knowl. Eng. **57**(2), 122–147 (2006)
19. Ouzzani, M., Bouguettaya, B.: Efficient access to Web services. IEEE Internet Computing **37**(3), 34–44 (2004)
20. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: SIGMOD (2003)
21. Pei, J., Jin, W., Ester, M., Tao, Y.: Catching the best views of skyline: a semantic approach based on decisive subspaces. In: VLDB (2005)

22. Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Springer, Berlin (1985)
23. Schmidt, C., Parashar, M.: A peer-to-peer approach to web service discovery. World Wide Web **7**(2), 211–229 (2004)
24. Skoutas, D., Sacharidis, D., Simitsis, A., Sellis, T.: Ranking and clustering web services using multicriteria dominance relationships. IEEE T. Service Computing **3**, 163–177 (2010)
25. Srivastava, D., Dar, S., Jagadish, H.V., Levy, A.Y.: Answering queries with aggregation using views. In: VLDB (1996)
26. Srivastava, U., Widom, J., Munagala, K., Motwani, R.: Query optimization over Web services. In: VLDB (2006)
27. Tan, K., Eng, P., Ooi, B.: Efficient progressive skyline computation. In: VLDB (2001)
28. Tao, Y., Xiao, X., Pei, J.: Subsky: efficient computation of skylines in subspaces. In: ICDE (2006)
29. Xu, Z., Martin, P., Powley, W., Zulkernine, F.: Reputation-enhanced qos-based web services discovery. In: ICWS, pp. 249–256 (2007)
30. Yu, Q., Bouguettaya, A.: Framework for Web service query algebra and optimization. ACM Trans. Web **2**(1), 1–35 (2008)
31. Yu, Q., Liu, X., Bouguettaya, A., Medjahed, B.: Deploying and managing web services: issues, solutions, and directions. VLDB J. **17**(3), 537–572 (2008)
32. Yu, Q., Bouguettaya, A.: Computing service skyline from uncertain qows. IEEE T. Service Computing **3**(1), 16–29 (2010)
33. Yu, T., Zhang, Y., Lin, K.-J.: Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans. Web **1**(1), 6 (2007)
34. Yuan, Y., Lin, X., Liu, Q., Wang, W., Yu, J., Zhang, Q.: Efficient computation of the skyline cube. In: VLDB (2005)
35. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.: Quality-driven Web service composition. In: WWW (2003)
36. Zhang, S., Mamoulis, N., Cheung, D.W.: Scalable skyline computation using object-based space partitioning. In: SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data, pp. 483–494. ACM, New York (2009)