Computing Service Skylines over Sets of Services

Qi Yu
College of Computing and Information Sciences
Rochester Institute of Technology, USA
Email: qi.yu@rit.edu

Athman Bouguettaya
CSIRO ICT Center
Canberra, ACT, Australia
Email: athman.bouguettaya@csiro.au

Abstract—We propose a skyline computation approach that enables service users to optimally access sets of services as an integrated service package. We first present a one pass algorithm based on the observation that a multi-service skyline is completely determined by single service skylines. The skyline is returned after an enumeration on a significantly reduced candidate space. We then develop a dual progressive algorithm that is able to progressively report the skyline. We conduct an experimental study to assess the performance of the skyline computation approaches.

I. INTRODUCTION

There has been a large steady flow of Web services deployed over the Web, largely thanks to the wide acceptance of SOA as the solution to interoperation, reuse, and globalization. As Web services with similar functionality are expected to be provided by competing providers, a major challenge is devising optimization strategies for finding the “best” Web services or compositions thereof with respect to the expected user-supplied quality.

Example 1: Suppose that a user wants to do a trip planning. Typical Web services that would need to be accessed include TripPlanner, Map, and Weather. TripPlanner provides basic trip information, such as airlines, hotels, and local attractions. Other than this, users may also be interested to consult the city map, local transportations, etc, by accessing the Map service. The weather condition during the travel days is an important factor that makes the Weather service relevant.

When accessing a Web service (e.g., Map in Example 1), most users may prefer selecting the service providers that best satisfy their desired quality requirement (e.g., fee, response time, and reputation, etc). However, this usually requires a series of trial-run processes and would be very painstaking if the number of competing providers is large. It may become more complicated if users need to access sets of services as an integrated service package. For example, to get a desired trip package, users may need to focus on the overall quality of the entire service package, such as the total response time, fee, and reputation. The possible combinations of providers for the service package are likely to exceed the range for a manual selection by the users. Therefore, disciplined optimization strategies are required for finding the “best” Web services or compositions thereof with respect to the Quality of Web services (QoWS).

A family of Objective Function based approaches (referred to as OF) has been studied for service optimization. OF predefines a score function \( \mathcal{F}(\vec{q}, \vec{w}) \), where \( \vec{q} \) is a set of quality of service parameters and \( \vec{w} \) is a set of weights assigned for each parameter in \( \vec{q} \). The score function \( \mathcal{F} \) assigns a scalar value to each service provider and the provider gaining the highest value will be selected and returned to the user. A major limitation affiliated with OF is that it requires users to transform personal preferences into numeric weights. Users may not have, for whatever reason, the ability to make tradeoff decision between different quality aspects using numbers. Furthermore, OF works like a “black box”, where users submit their weights over quality parameters and OF returns the system selected provider. Users thus lose the flexibility to “visualize” and select their desired providers.

Computing the skylines representing the “best” QoWS along each dimension (referred as service skylines) comes as a natural solution that addresses the issues of using OF. Skyline computation has received significant consideration in the database community. For a \( d \)-dimensional data set, the skyline consists of a set of points which are not dominated by any other points. A point \( \vec{p} = (p_1, ..., p_d) \) dominates another point \( \vec{r} = (r_1, ..., r_d) \) if \( \forall i \in [1, d], p_i \geq r_i \) and \( \exists j \in [1, d], p_j > r_j \). We use \( \geq \) 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 that are not dominated by others in terms of all user interested QoWS aspects, such as response time, fee, and reputation. Computing service skylines can completely free users from assigning weights over different QoS parameters. The skylines also guarantee that the user desired service providers are included so that users can make flexible selection from them.

In this paper, we focus on the challenging problem of computing service skylines over sets of services. Using Example 1 a complex service package (like a trip package) is formed by combining services from different providers (e.g., TripPlanner, Map, and Weather). The possible combinations of service providers will increase exponentially with the number of services involved. Suppose that there are \( n_1 \), ..., \( n_m \) providers for each of the \( m \) services in a service package. To find the skyline for the service package, \( \prod_{i=1}^{m} n_i \) number of candidates need to be considered. The computational cost would be prohibitive if any one of the following conditions is true: \( (C_1) \) the number of services in a service package is large,
An intuitive solution that addresses the above challenge is to compute the multi-service skylines from single service skylines. This solution is valid due to two observations: \((O_1)\) a multi-service skyline point can only be formed by a set of single service skyline points; \((O_2)\) the sizes of skylines are usually much smaller than the sizes of the service space (i.e., the number of providers). In this case, only \(N = \prod_{i=1}^{m} k_i\) maximum number of candidates need to be evaluated due to \(O_1\), where \(k_i\) is the size of the \(i\)th service skyline and we expect that \(k_i \ll n_i\) due to \(O_2\). Thus, the computational cost can be reduced with several orders of magnitude. However, \(N\) can still be large as more services are involved in the service package. In this paper, we develop a set of algorithms to efficiently compute multi-service skylines. The major contributions are summarized as follows:

- We first develop a baseline algorithm (referred to as One Pass Algorithm or OPA), which performs a single pass on the \(N\) candidates and outputs the skyline. It employs a special enumeration mechanism to effectively reduce the number of false positive skyline points during skyline computation.
- We present a Dual Progressive Algorithm (referred to as DPA) that is able to progressively report the skyline. It leverages an expansion lattice and a parent table to ensure the efficiency and progressiveness.
- We perform a set of experiments to evaluate the proposed multi-service skyline computation algorithms.

We provide the formal definition of the multi-service skyline computation problem in Sect. III. We present the multi-service skyline computation algorithms in Sect. \(\text{III}\). We experimentally evaluate and compare the proposed algorithms under different settings in Sect. \(\text{IV}\). We discuss related work in Sect. \(\text{V}\), and provide some concluding remarks in Sect. \(\text{VI}\).

**II. PRELIMINARIES**

In this section, we first provide the formal definition of service skylines. We then illustrate the challenges of computing skylines over sets of services. We then formally define the multi-service skyline computation problem. A set of notations is summarized in Table \(\text{I}\) for reference.

A. Service Skylines

Before giving the formal definition of service skylines, we first introduce the concept of Service Execution Plans (SEPs) \([13]\). Accessing a Web service typically includes the invocation of a set of operations. For example, accessing a Map service requires to invoke two operations: Geocode and GetMap. There may be dependency constraints between these operations (e.g., GetMap depends on Geocode). A SEP arranges these operations into a sequence with respect to the dependency constraints. Since there may be multiple providers competing to offer the same service, multiple SEPs can be generated from these providers. The quality parameters of a SEP are computed by aggregating those of its member service operations. These aggregation functions have been widely used in various service optimization problems \([13], [16], [13]\).

Table \(\text{I}\) shows these aggregation functions. The quality of a SEP is represented as a vector, \((\text{lat, rel, avail, fee, rep})\).

**TABLE I**

<table>
<thead>
<tr>
<th>QoWS parameter</th>
<th>Aggregation function</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency</td>
<td>(\sum_{i=1}^{m} \text{latency}(\text{op}_i))</td>
</tr>
<tr>
<td>reliability</td>
<td>(\sum_{i=1}^{m} \log(\text{reliability}(\text{op}_i)))</td>
</tr>
<tr>
<td>availability</td>
<td>(\sum_{i=1}^{m} \log(\text{availability}(\text{op}_i)))</td>
</tr>
<tr>
<td>fee</td>
<td>(\sum_{i=1}^{m} \text{fee}(\text{op}_i))</td>
</tr>
<tr>
<td>reputation</td>
<td>(\sum_{i=1}^{m} \text{reputation}(\text{op}_i))</td>
</tr>
</tbody>
</table>

**Example 2:** Using our running example to illustrate the above concepts, assume that a user wants to access the city map. The service consists of two operations: (Geocode, GetMap). Assume that there are three service providers, \(A, B,\) and \(C\) that offer this Map service. The QoS of these providers is given in Figure \(\text{I}\). It is also worth to note that the QoS attributes are usually divided into two categories: positive and negative. In negative (resp. positive) attribute, the higher (resp. lower) the value, the worse is the quality. Among the quality parameters we considered in our running example, latency and fee are negative attributes while the others are positive attributes. These three providers will result in three SEPs: SEP\(_A\), SEP\(_B\), and SEP\(_C\). Based on the aggregation functions defined in Table \(\text{I}\) we compute the QoS of these SEPs as: \((1.5, -.09, -.19, 0, 4), (2.7, -.19, -.31, 0.8, 3),\) and \((2.0, -.09, -.31, 0.7, 4)\).
SEP_A ⊳ SEP_C. Since SEP_A is not dominated by any other SEPs, it is in the service skyline.

For each service request, a set of SEPs is dynamically generated. As a result, SEP skyline computation is more challenging than database skyline computation. When computing database skylines, the database remains static. Database indexing structures are usually constructed in advance to expedite the skyline computation. One way to compute the SEP skyline is to leverage existing non-index based skyline algorithms (e.g., [1], [2], [3]). The procedure is briefly described as follows. When the SEPs are generated for a given request, the QoWS values are computed based on the aggregation functions given in Table I. The SEP’s performance is then just represented as a vector. Algorithms like BNL [1] or SFS [2] can be applied to compute the service skylines.

B. Problem Definition

As more services are involved into a service package, the number of SEPs is expected to increase exponentially [13]. Let us consider a service trip package with two services: TripPlanner and Map. Four operations, TripSearch, GetTrip, Geocode, GetMap, are involved in this package. To generate the final SEPs, we first generate SEPs for each single service and then combine them. The TripPlanner service is offered by two providers M and N and the Map service is offered by three providers A, B, and C. Therefore, we have two SEPs (i.e., SEP_M and SEP_N) for the TripPlanner service and three SEPs (i.e., SEP_A, SEP_B, and SEP_C) for the Map service. Since there are six possible combinations between these two groups of SEPs, six SEPs are generated: SEP_M, SEP_A, SEP_B, SEP_M, SEP_C, and SEP_N. The QoWS of these final SEPs is computed by aggregating the QoWS of the single service SEPs based on functions in Table I. For example, the QoWS of SEP_A is (1.5, -0.9, -1.9, 0, 4) and the QoWS of SEP_M is (4.0, -3.1, -1.9, 0, 4). Therefore, the QoWS of SEP_M is (5.5, -4.4, -3.8, 0, 4).

A complex service package (e.g., a travel package) typically requires access to multiple services where each service may be offered by a large number of providers. Assume that there are \( n_1, \ldots, n_m \) providers for each of the \( m \) services in a service package, \( \prod_{i=1}^{m} n_i \) number of SEPs may need to be considered. A straightforward extension on approaches, such as BNL and SFS, may not provide efficient solutions for computing multi-service skylines.

In this paper, we present techniques that efficiently compute service skylines over sets of services. We define a set of notions that are used to illustrate these techniques. Other terminologies used throughout the paper are summarized in Table II.

- For clarity, we use the term SEP to specifically refer to a service execution plan for a single service. The SEP skyline is used to refer to the skyline computed from the SEPs.
- We use the term MSEP (i.e., Multi-Service Execution Plan) to refer to a service execution plan for sets of services. For example, following this notion, since SEP_AM is a multi-service execution plan, it can be denoted as MSEP(A, M).
- Another important concept used in our approach is the scores of SEPs and MSEPs. We provide the formal definitions as follows.

**Definition 3: (Score of a SEP).** The score of a SEP is computed as

\[
\sum_{i=1}^{k} f^{(1)}_i \times (\max(q_i) - q_i) + f^{(2)}_i \times q_i
\]

where \( q_i \) is a QoWS attribute given in Table II and \( k \) is the number of QoWS attributes. \( f^{(1)}_i = 1, f^{(2)}_i = 0 \) if \( q_i \) is a positive attribute and \( f^{(1)}_i = 0, f^{(2)}_i = 1 \) if \( q_i \) is a negative attribute. \( \max(q_i) \) is computed by considering all the SEPs for the same service.

Consider the three SEPs in Example 2. For the three positive QoS attributes, we have \( \max(availability) = -0.09, \max(reliability) = -0.19 \) and \( \max(reputation) = 4 \). For example, the score of SEP_A can be computed as follows: score(SEP_A) = 1.5 + (-0.09 - (-0.09)) + (-0.19 - (-0.19)) + (4 - 4) = 1.5.

**Definition 4: (Score of a MSEP).** The score of a MSEP is the sum of the scores of its SEPs.

For example, MSEP(A, M) consists of two SEPs: SEP_A and SEP_M. The score of SEP_M can be computed as the same way as that of SEP_A. Thus, score(MSEP(A,M))=score(SEP_A) + score(SEP_M) = 1.5 + 5.12 = 6.62. Based on the above score definitions, we have the following observation.

**Property 1:** If SEP_1 ⊳ SEP_2 (resp. MSEP_1 ⊳ MSEP_2), then score(SEP_1) < score(SEP_2) (resp. score(MSEP_1) < score(MSEP_2)).

**Proof:** The score definitions essentially have the effect that the better the quality the less the score. SEP_1 ⊳ SEP_2 means that SEP_1 is better than SEP_2 in all QoWS aspects, thus score(SEP_1) < score(SEP_2).

**Problem Definition.** Given \( m \) services with \( d \) user interested QoS attributes, where the \( i \)th service has \( n_i \) different providers, the problem of MSEP skyline computation is to compute the service skyline \( MS \) over the \( m \) services.
TABLE II
TERMINOLOGIES

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEP</td>
<td>service execution plan for a single service</td>
</tr>
<tr>
<td>MSEP</td>
<td>service execution plan for sets of services</td>
</tr>
<tr>
<td>SKi</td>
<td>the ith SEP skyline</td>
</tr>
<tr>
<td>ki</td>
<td>size of the ith SEP skyline</td>
</tr>
<tr>
<td>score</td>
<td>the score of a SEP or MSEP</td>
</tr>
<tr>
<td>MS</td>
<td>the MSEP skyline</td>
</tr>
<tr>
<td>A&gt;B</td>
<td>A dominates B</td>
</tr>
<tr>
<td>N</td>
<td>size of the MSEP space</td>
</tr>
<tr>
<td>m</td>
<td>number of services</td>
</tr>
<tr>
<td>d</td>
<td>number of QoWS attributes</td>
</tr>
</tbody>
</table>

III. COMPUTING SERVICE SKYLINES OVER SETS OF SERVICES

Computing the MSEP skyline in a brute force manner is computationally intensive. The following observation helps improve the computation efficiency directly.

**Lemma 1:** Given $m$ services $S_1, ..., S_m$ and the set of SEP skylines $SK_1, ..., SK_m$, computed for each of them, the MSEP skyline $MS$ over $S_1, ..., S_m$ can be completely decided by $SK_1, ..., SK_m$.

**Proof:** For a MSEP, $\psi \in MS$, assume that it consists of $m$ SEPs, $SEP_1, ..., SEP_m$, one from each service. Assume that all SEPs are from the corresponding SEP skylines respectively, except for $SEP_j$, such that $SEP_j \notin SK_j$. Thus, there must be a $SEP_j' \in SK_j$ such that $SEP_j' \triangleright SEP_j$. Therefore, we can find a MSEP $\psi'$ by replacing $SEP_j$ in $\psi$ with $SEP_j'$ such that $\psi' \triangleright \psi$. This contradicts the fact that $\psi$ is a skyline MSEP.

Lemma 1 enables us to compute the MSEP skylines by only considering the SEP skylines. Based on this lemma, we present two algorithms to compute the MSEP skylines. The **One Pass Algorithm** (OPA) performs a single pass on the MSEP space with a size of $N = \prod_{i=1}^{m} k_i$ to compute the MSEP skyline. The **Dual Progressive Algorithm** (DPA) leverages an expansion lattice and a heap to progressively compute the skyline.

A. One Pass Algorithm

We present the OPA algorithm in this section. During the single pass of the MSEP space, OPA enumerates the candidate MSEPs one by one and only stores the potential skyline MSEPs. It outputs the skyline after all the candidate MSEPs have been evaluated. OPA requires that all the SEP skylines are sorted according to the scores of the SEPs. OPA works as follows (shown in Algorithm 1). It starts by evaluating the first MSEP (referred to as MSEP$_1$) that is formed by combining the top SEPs from each SEP skyline. It is guaranteed that MSEP$_1 \in MS$ because MSEP$_1$ has the minimum score in the MSEP space so that no other MSEPs can dominate it. With the minimum score, MSEP$_1$ is expected to have a very good pruning capacity. Thus, OPA puts MSEP$_1$ on the top of $MS$ so that the non-skyline MSEPs which are dominated by MSEP$_1$ can be pruned at the earliest time. After this, OPA continues to enumerate all other MSEPs, one by each time. For each MSEP$_i$, the algorithm checks it against the current $MS$. When MSEP$_i$ meets the first MSEP, say MSEP$_j$, that can dominate it, the algorithm prunes MSEP$_i$, and stops further checking. On the other hand, if MSEP$_i$ dominates any MSEP, say MSEP$_j$, in $MS$, MSEP$_j$ will be removed from the skyline. If none of the MSEPs in $MS$ can dominate MSEP$_i$, MSEP$_i$ will be inserted into the skyline.

One outstanding issue with OPA is the false positive skyline MSEPs, which incur additional space and CPU cost. The false positives are generated because OPA has no restriction on the enumeration order of the MSEPs. Some early discovered MSEP that has been inserted into the skyline may be dominated by other later discovered MSEPs. These false positive MSEPs will stay in the skyline (which introduces space cost) and be compared with all the MSEPs discovered after them until being dominated (which introduces CPU cost).

To reduce the number of false positives, we add some special control on the EnumerateNext function of OPA. Suppose that there are $m$ SEP skylines (each of them is sorted on the scores of its SEPs). EnumerateNext first returns MSEP$_1$. It then keeps increasing the index of the $m$th SEP skyline to enumerate the remaining MSEPs. When the index hits the end of the $m$th SEP skyline, the index of $(m-1)$th SEP skyline will increase by 1 and the index of the $m$th skyline will reset to 0. This will propagate to all other SEP skylines until all MSEPs are enumerated. Since all the SEP skylines are sorted, this process tends to enumerate the MSEPs by roughly following the ascending order of their scores. The effectiveness of the EnumerateNext function is justified by our experimental results. It needs to be emphasized here that in order to make the EnumerateNext effective, the skylines need to be originally sorted.

Algorithm 1 One Pass Algorithm

**Input:** $m$ sorted SEP skylines $SK_1, ..., SK_m$

**Output:** The MSEP skyline $MS$

1: $N = \prod_{i=1}^{m} |SK_i|$; // number of candidate MSEPs
2: for all $i \in \{2, N\}$ do
3:   MSEP$_i$ = EnumerateNext(SK$_1$, ..., SK$_m$);
4:   IsDirected = False;
5:   for all $j \in \{1, |MS|\}$ do
6:     MSEP$_j$ = $MS$.get(j);
7:     if MSEP$_i$ \triangleright MSEP$_j$ then
8:       $MS$.remove(j);
9:   else if MSEP$_j$ \triangleright MSEP$_i$ then
10:      IsDirected = True;
11:     break;
12: end if
13: end for
14: if IsDirected == False then
15:   $MS$.add(MSEP$_i$);
16: end if
17: end for

B. Dual Progressive Algorithm

We present the DPA algorithm in this section. The underlying principle of DPA is the dual progressive strategy, which we briefly elaborate as follows.
DPA **progressively** enumerates the MSEPs in an ascending order of their scores. By “progressively”, we mean that it does not need any presorting on the MSEPs, which is usually very time consuming and thus blocks on input \(2, 3\). DPA “intelligently” retrieves the MSEPs based on their scores one at a time. This is non-trivial if the entire MSEP space is not initially sorted. DPA follows the enumeration steps defined by an **expansion lattice** and leverages a set of key data structures to achieve this, which will be explained in what follows.

**DPA progressively** reports the skyline MSEPs. By “progressively”, we mean that once a MSEP is discovered not being dominated by any existing skyline MSEP, it is guaranteed to be in the skyline and can be returned to the user. This also implies that no false positives are generated by DPA. This is because that a later discovered MSEP cannot have a smaller score than an earlier discovered MSEP due to the first progressive property of DPA. Based on the definition of score, it is impossible for MSEP \(_j\) to dominate MSEP \(_i\), because otherwise the former should have a smaller score than the latter.

The dual progressive property helps DPA gain two major advantages: (1) DPA is a completely pipelineable algorithm, i.e., it blocks on neither the input nor the output. It does not block on input because presorting is no longer needed. Meanwhile, the progressive generation of the skyline makes it non-block on output. (2) Since only true skyline MSEPs are kept in \(MSEP\), the space overhead is greatly reduced. In addition, many needless comparisons are expected to be eliminated.

1) **Basic Progressive Enumeration:** The second progressive property is essentially a natural result from the first progressive property. We focus on investigating how the first progressive property is achieved in this section. Similarly to OPA, DPA also requires that the SEP skylines are all sorted. The entire MSEP space can be enumerated systematically using a MSEP expansion lattice. Figure 3 shows the expansion lattice for three SEP skylines, \(A(a_1, a_2, a_3), B(b_1, b_2, b_3)\), and \(C(c_1, c_2, c_3)\). The number of MSEPs generated from these skylines will be \(|A| \times |B| \times |C| = 27\). Each node of the expansion lattice corresponds to a MSEP. In particular, the root node (referred to as \(r\) or \(n_1\)) corresponds to \(MSEP_1\), i.e., the MSEP that is formed by the top SEPs from each SEP skyline. \(MSEP_1\) has the smallest score and thus must belong to the skyline. A child node is different from a parent node by only one SEP and the SEP from the child node is the successor of the SEP from the parent in the corresponding SEP skyline.

It is worth to note that the MSEP expansion lattice is completely different from the Low-Cardinality Lattice (LCL) used by [6] for computing skylines over low-cardinality domains. In LCL, a node, say \(n_i\) is always dominated by its ancestors. Therefore, as long as one of its ancestor is present in the data space, \(n_i\) can be removed. However, in the MSEP expansion lattice, we only know that \(n_i\) has a larger score than its ancestors, which does not mean that \(n_i\) is dominated by its ancestors. Therefore, the MSEP expansion lattice only defines the enumeration order between the nodes and make sure the node \(n_i\) is enumerated after its ancestors. For nodes that do not have an ancestor-successor relationship (e.g., \((a_2, b_1, c_1)\) at level 1 and \((a_1, b_3, c_1)\) at level 2), we should also ensure a correct enumeration order (because \((a_1, b_3, c_1)\) may have a smaller score than \((a_2, b_1, c_1)\) and thus should be enumerated earlier).

Specifically, we use the expansion lattice \(T\) together with a heap \(\mathcal{H}\) to achieve the basic progressive enumeration. The expansion lattice ensures that a parent node is enumerated before its child nodes. This is desirable because the score of a parent node cannot be larger than those from its children. The heap, on the other hand, determines the enumeration order of nodes that do not have a parent-child relationship. The enumeration starts by initializing the heap \(\mathcal{H}\) with \(MSEP_1\) (i.e., \(n_1\)). Each enumeration step consists of two sub-steps: (1) **Extract**—the MSEP with the smallest score, say \(n_i\), is extracted from \(\mathcal{H}\) and compared with the existing skyline. \(n_i\) will be inserted into the skyline if not dominated and discarded if otherwise. (2) **Expand**—the child nodes of \(n_i\) are then generated and inserted into \(\mathcal{H}\). The enumeration stops when \(\mathcal{H}\) is empty.

2) **Node Duplication:** A major issue with the above basic implementation is that a single node could be generated from parent expansion for multiple times, referred to as **node duplication**. This is because that a node could have up to \(m\) parents, where \(m\) is the number of SEP skylines. The same child node will be generated when each of its parents is expanded. As shown in Figure 6 the number above each node indicates the number of its parents. For example, the node \((a_2, b_2, c_2)\) will be inserted into \(\mathcal{H}\) for three times because it has three parents and when each of them is expanded, \((a_2, b_2, c_2)\) will be generated and inserted into \(\mathcal{H}\). The node duplication issue introduces great computational overhead because lots of node are processed multiple times. More seriously, the same node could be inserted into the skyline for more than one time, which results in a wrong skyline.

A straightforward extension to tackle the above issue is to add a **Parent Checking** (PC) procedure before inserting a node into the heap. For a given node, say \(n_i\), generated
from the expansion, PC examines whether there is any of its parents currently in the heap. If this is the case, the child will not be inserted. However, PC still cannot completely resolve the above issue. We use a specific example to explain this. With the expansion tree shown in Figure 3 nodes \((a_2, b_1, c_1), (a_1, b_2, c_1), \) and \((a_1, b_1, c_2)\) will be generated and inserted into \(H\) after \((a_1, b_1, c_1)\) is expanded. Assume that \((a_2, b_1, c_1)\) has the smallest score so that it will be extracted from \(H\). After \((a_2, b_1, c_1)\) is expanded, its three child nodes \((a_3, b_1, c_1), (a_2, b_2, c_1)\), and \((a_2, b_1, c_2)\) are generated. With the checking procedure, only \((a_3, b_1, c_1)\) is inserted into \(H\) because the parents of \((a_2, b_2, c_1)\), and \((a_2, b_1, c_2)\) are still in \(H\). Assume that \((a_3, b_1, c_1)\) has the smallest score at this point. It is then extracted from \(H\) and expanded to generate its two child nodes \((a_3, b_2, c_1)\) and \((a_3, b_1, c_2)\). With the checking procedure, both \((a_3, b_2, c_1)\) and \((a_3, b_1, c_2)\) are inserted into \(H\) since none of their parents are in the heap. This is problematic because \((a_3, b_2, c_1)\) and \((a_3, b_1, c_2)\) are placed into the heap before some of their parents, i.e., \((a_2, b_2, c_1)\) and \((a_2, b_1, c_2)\). In this case, when the parent nodes are expanded, \((a_3, b_2, c_1)\) and \((a_3, b_1, c_2)\) will be generated again, respectively. This, again, results in node duplication.

The above analysis reveals that in order to completely avoid node duplication, we should check whether there is any ancestor (instead of only parent) of a node currently in the heap, i.e., as long as any ancestor of a node is still in the heap, the node will not be inserted into the heap. In the above example, \((a_3, b_2, c_1)\) and \((a_3, b_1, c_2)\) will not be inserted into \(H\) with ancestor checking because their ancestors \((a_1, b_2, c_1)\) and \((a_1, b_1, c_2)\) are in the heap. Node \(n_j\), represented as \((SEP_{1j},...,SEP_{mj})\) is an ancestor of node \(n_i\), represented as \((SEP_{1i},...,SEP_{mi})\) if \(SEP_{kj}.score \leq SEP_{bi}.score, 1 \leq k \leq m\). Therefore, the complexity of checking an ancestor relationship is \(\Theta(m)\). For each node, the entire \(H\) needs to be checked. Thus, ancestor checking for a node requires a complexity of \(\Theta(|H| \times m)\), where \(|H|\) is the length of the heap at the point when the checking is conducted. Since a node will be generated when each of its parent is expanded, the overall complexity of ancestor checking for each node is \(\Theta(|H| \times m \times p)\), where \(p\) is the number of parents for the node. This is expensive especially when the size of \(H\) becomes large with the increase of the number of services.

3) Parent Table: We introduce the parent table data structure in this section. The parent table provides a decent solution to tackle the node duplication issue with minimum overhead. Instead of keeping track of all the ancestors, the parent table only stores the information related to the number of parents for a given node. The underlying principle is that a node can be inserted into the heap only after all its parents have been processed. Since the maximum number of parents for a node is \(m\), with \(m\) as the number of SEP skylines, the parent table only uses up to \((\log m + 1)\) bits for a given node. Now the question is how to decide the number of parents for a node. Based on the expansion lattice, we have the following property.

**PROPERTY 2:** Assume that the index of each SEP skyline starts with 1. The number of parents for a given node \(n_i\), represented as \((SEP_{1i},...,SEP_{mi})\), equals to the number of SEPs with an index greater than 1.

Figure 4 shows the number of parents for each node in the expansion lattice. With the parent table, the progressive enumeration now works as follows. The parent table, referred to as \(P\), is first initialized by setting each node entry as the number of parents for the node (refer to Property 2). Similar to the basic implementation, the heap \(H\) is initialized with the root of the expansion lattice \(T\). Each enumeration step now consists of three sub-steps. The Extract and Expand are the same as before except that Expand only generates the child nodes and does not insert any of them into \(H\). A new UpdateCheck step is added, which works as follows. For each newly generated child node, it first updates \(P\) by subtracting 1 from the corresponding node entry. The updated entry now represents the remaining parent nodes that have not been processed yet. It then checks the entry. If the entry becomes 0, the corresponding node will be inserted into \(H\). By doing this, we make sure that a child node can only be inserted into the heap after all its parent nodes have been processed. Each UpdateCheck takes a complexity of \(\Theta(1)\). Thus, for a node that has \(p\) parents, the overall overhead is \(\Theta(p)\).

The detailed progressive enumeration algorithm (referred to as PEN) is given in Algorithm 2.

```
Algorithm 2 Progressive ENumeration (PEN)
Input: \(m\) SEP skylines that form an expansion lattice \(T\)
Output: The MSEP skyline \(MS\)
1: \(MS = \phi, H = \phi;\)
2: Initialize the parent table \(P;\)
3: while \(H \neq \phi\) do
4: remove the top node \(n\) from \(H;\)
5: if \(n\) is not dominated by any node in \(MS\) then
6: \(MS.add(n);\)
7: end if
8: \(CN = expand(n, T); // generate the child nodes\)
9: for all node \(n_i \in CN\) do
10: \(P(n_i) = P(n_i) - 1;\)
11: if \(P(n_i) = 0\) then
12: \(H.add(n_i);\)
13: end if
14: end for
15: end while
```

Example 4: We use an example as shown in Figure 4 to explain the progressive enumeration process. Figure 4(c) shows the enumeration steps with Figure 4(d) and (e) illustrating the contents of the parent table and the heap, respectively. In step 1, \((a_1, b_1)\) is removed from the heap and expanded to \((a_2, b_1)\) and \((a_1, b_2)\). For each of these two nodes, the corresponding entries in the parent table are first updated, i.e., \(P(a_1, b_2) = P(a_1, b_2) - 1 = 0, P(a_2, b_1) = P(a_2, b_1) - 1 = 0\). Since both entries become 0 after the update, \((a_2, b_1)\) and \((a_1, b_2)\) are inserted into the heap. In step 2, \((a_1, b_2)\) will be removed from the heap since it has the smallest score. It is expanded to MSEP\((a_2, b_2)\) and MSEP\((a_1, b_3)\). The parent
Table III gives the parameter settings of the experiments.

**IV. EXPERIMENTAL STUDY**

We implement both of the proposed algorithms: OPA and DPA. We conduct a set of experiments to assess the performance of these algorithms. The QoWS attributes of service instances are generated in two different ways following the approach described in [1]: 1) *Independent QoWS* where all the QoWS attributes of service instances are uniformly distributed, and 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. We first compute the SEP skylines and then sort the SEP skylines based on the scores of their SEPs.

**TABLE III**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>Cardinality of service relations</td>
<td>(100k)</td>
</tr>
<tr>
<td>(d)</td>
<td>Number of QoWS attributes</td>
<td>[2, 5])</td>
</tr>
<tr>
<td>(m)</td>
<td>Number of Services</td>
<td>[2, 4])</td>
</tr>
</tbody>
</table>

We evaluate the performance in terms of total CPU cost against \(d\) and \(m\) in this section. It is worth to note that the sizes of the SEP skylines keep increasing with \(d\), which is especially obvious with the anti-correlated QoWS. For instance, the size of a SEP skyline typically exceeds 1000 when \(d \geq 5\). In practice, a service query may typically pose some other quality constraints, which help prune a large portion of the SEP skyline. To avoid overly large SEP skylines, we select the top 100 skyline SEPs (based on their scores) from them. Under this setting, the candidate MSEP space \(\mathcal{N}\) can still become very large with the increase of the number of services, for example, \(\mathcal{N} = 10^8\), when \(m = 4\). Considering the potentially large service space, manually selecting an ideal MSEP is actually impossible. The proposed service skyline computation algorithms provide systematic way that can help users efficiently select their desired MSEPs.

**Fig. 4.** Progressive Enumeration

**Fig. 5.** CPU Time Vs. \(d\) (\(m = 4\))

**Fig. 6.** CPU Time Vs. \(m\) (\(d = 3\))

Figure 5 compares the two algorithms, BUA and DPA, with \(m = 4\) and \(d \in [2, 5]\). OPA computes the whole service skyline more efficiently than DPA. This is mainly because that DPA requires additional overhead (e.g., using a heap) to maintain the progressiveness when reporting the skyline. OPA also benefits from the effectiveness of the *EnumerateNext* function, which is demonstrated by Figure 6.

Figure 6 evaluates the impact of \(m\). The results demonstrate a similar trend. One additional observation is that the performance difference between these two algorithms become more obvious as \(m\) increases. This is because the the heap size \(|H|\) increases quickly with \(m\), which increases the overhead for DPA.

Figure 7 evaluates how the sizes of the MSEP skylines vary in OPA, which also helps demonstrate the effectiveness of the *EnumerateNext* function. The sizes of MSEP skylines...
general keep increasing over all iterations. The sizes only drop occasionally and slightly due to the removing of the false positives. Therefore, the EnumerateNext function effectively reduces the chances of generating false positive skyline MSEP s, which accounts for efficient performance of OPA for generating the entire MSEP skylines.

V. RELATED WORK

Query optimization techniques have been developed for service computing recently. In [16], a Web Service Management System (WSMS) is proposed to enable optimized querying over Web services. An algorithm is proposed to arrange service calls into a pipelined execution plan. In [7], 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. A service query can be transformed over these levels to generate an optimal execution plan. In [10], a composite service optimization approach is proposed based on several quality of service parameters. The optimization problem is tackled by finding the best Web services to execute a composite service in the form of a linear programming problem. We propose in this paper the skyline computation approach to address the service selection problem. The service skyline algorithms are developed upon a formal service query framework, which goes beyond the ad hoc query model [7]. The focus of service skyline computation is not only to efficiently answer the service query [10]. It also guarantees that optimality on the quality of the query result. Finally, computing the service skylines offers some inherent advantages over the objective function based approaches [16], [13], [15], including less user intervention and increased flexibility for users to select desired service providers.

Skyline computation has been extensively investigated in the database community [1], [12], [5], [3], [2], [8]. Some techniques can be employed to compute single service skylines (i.e., SEP skylines). Jin et al. investigated the skyline operator on multi-relational databases. The focus is on integrating efficient join methods into skyline computation based on the Primary Key and Foreign Key (PK-FK) relationship [4]. Sun et al. studied a similar problem in the distributed environment [11]. The multi-service skyline algorithms proposed in this paper essentially assume that a Cartesian product is performed among multiple services. Therefore, no PK-FK relationship can be leveraged to prune the searching space. More challengingly, Cartesian product typically results in a much larger number of execution plans. The proposed algorithms can efficiently and progressively report the service skyline.

VI. CONCLUSION

We proposed a skyline computation approach for service selection. The obtained skyline, called multi-service skyline, enables service users to optimally and efficiently access sets of services as an integrated service package. We developed two algorithms that include a baseline one pass algorithm, OPA and a dual progressive algorithm, DPA. DPA employs an expansion lattice and a parent table to achieve progressiveness. The multi-service skyline algorithms provide key extensions to the existing skyline approaches. They allow efficient computation of service skylines over sets of services. An interesting future direction is to investigate ways to improve the overall performance of DPA while allowing it to progressively generate the skyline.