Efficient Service Skyline Computation for Composite Service Selection

Qi Yu, *Member*, *IEEE*, and Athman Bouguettaya, *Fellow*, *IEEE*

Abstract—Service composition is emerging as an effective vehicle for integrating existing web services to create value-added and personalized composite services. As web services with similar functionality are expected to be provided by competing providers, a key challenge is to find the "best" web services to participate in the composition. When multiple quality aspects (e.g., response time, fee, etc.) are considered, a weighting mechanism is usually adopted by most existing approaches, which requires users to specify their preferences as numeric values. We propose to exploit the *dominance relationship* among service providers to find a set of "best" possible composite services, referred to as a *composite service skyline*. We develop efficient algorithms that allow us to find the composite service skyline from a significantly reduced searching space instead of considering all possible service compositions. We propose a novel bottom-up computation framework that enables the skyline algorithm to scale well with the number of services in a composition. We conduct a comprehensive analytical and experimental study to evaluate the effectiveness, efficiency, and scalability of the composite skyline computation approaches.

Index Terms—Service composition, skyline, dominance analysis, quality of service

1 Introduction

One of the fundamental objectives of service computing is to enable interoperability among different software and data applications running on a variety of platforms. The introduction of web services has been key for the paradigm shift in business structures allowing them to outsource required functionality from third party web-based providers through service composition [21], [35]. The fast growing number of web services will result in a significant number of web services with similar functionalities. These competing services also pose a new challenge for service composition: selecting proper service providers that achieve a composition with the best user desired Quality of Web Service (QoWS).

A number of service selection approaches have been developed that depend on the computation of a predefined objective function [34], [31], [30], [23], [35]. A weighting mechanism is leveraged where users express their preference over different (and sometimes conflicting) quality parameters as numeric *weights*. The composition gaining the highest value from the objective function will be selected and returned to the user. There are two major limitations with these approaches. First, it is a rather demanding task for users to transform their personal preferences into numeric weights. Without a detailed knowledge about the QoWS from all possible compositions, users may not have the ability to make a precise tradeoff decision between different quality aspects using numbers. Second, whenever the weights are changed, a completely new search needs to be performed. An exhaustive search would be computationally expensive because the possible compositions will increase exponentially with the number of services involved [31].

1.1 Motivation

We propose to exploit the *dominance relationship* among service providers to find a set of "best" possible service compositions, referred to as a *composite service skyline*. We will use an example to motivate the key ideas.

Example 1 (Composing services). Consider the development of a composite web service, TravelAssistant, which provides travel assistance services for users. 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 these, users may also be interested to consult the city map and local transportations by accessing the Map service. The weather condition during the travel days is an important factor that makes the Weather service relevant. The developer may face a number of options for each of these services as there are multiple software vendors competing to offer similar functionalities. For example, Table 1 shows the five possible Map service providers and four possible TripPlanner providers.

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). Thus, these operations can be arranged into a sequence with respect to the dependency constraints, which is referred to as a *Service*

- *Q. Yu is with the College of Computing and Information Sciences, Rochester Institute of Technology, 152 Lomb Memorial Drive, Rochester, NY 14623–5608. E-mail: qi.yu@rit.edu.*
- *A. Bouguettaya is with the School of Computer Science and Information Technology, RMIT University, GPO Box 2476, Melbourne, VIC 30001, Australia. E-mail: athman.bouguettaya@rmit.edu.au.*

Manuscript received 2 Sept. 2011; revised 30 Nov. 2011; accepted 7 Dec. 2011; published online 16 Dec. 2011.
Recommended for acceptance by J. Yang.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2011-09-0529.
Digital Object Identifier no. 10.1109/TKDE.2011.268.

TABLE 1
Map and TripPlanner Providers

sid	Operation	Latency	Fee	Reputation
Map Providers				
A	GeoCode	0.5	0.8	2
	GetMap	1	0	2
B	GeoCode	0.7	0.3	3
	GetMap	2	0.5	3
C	GeoCode	0.5	0.2	2
	GetMap	1.5	0.9	2
D	GeoCode	0.3	0.7	2
	GetMap	1	0.4	2
E	GeoCode	0.6	0.7	3
	GetMap	0.8	0.5	3
TripPlaner Providers				
F	SearchTrip	2	0	2
	GetTrip	2	0.8	2
G	SearchTrip	1	0	3
	GetTrip	2	1	1
H	SearchTrip	2	1	2
	GetTrip	3	1	2
I	SearchTrip	3	1	3
	GetTrip	2	0	2

Execution Plan (SEP). The QoWS of a SEP are computed by aggregating those of its member service operations using a set of predefined aggregation functions [35], [31].

Example 2 (Service dominance). Consider three QoWS parameters, latency, fee, and reputation, which capture the response time, monetary cost, and users' rating of a service. Assume that a rating takes values in $[1, 5]$ and a smaller value reflects a better rating. Latency and fee of a SEP are computed as the sum of those from its member operations. The reputation is set as the average of those from the member operations. The QoWS of the SEPs for the five possible map providers are (1.5, 0.8, 2), (2.7, 0.8, 3), (2, 1.1, 2), (1.3, 1.1, 2), and (1.4, 1.2, 3). For two providers p_1 and p_2, if SEP_1 is as good as SEP_2 in all QoWS aspects and better than SEP_2 in at least one QoWS aspect, then SEP_1 ***dominates*** SEP_2, or denoted as $\text{SEP}_1 \prec \text{SEP}_2$. Therefore, SEP_A dominates both SEP_B and SEP_C, and SEP_D dominates SEP_E.

Since SEP_A and SEP_D are not dominated by any other providers, it is said that they are in the map *service skyline*. On the other hand, since SEP_B, SEP_C, and SEP_E are dominated by SEP_A and SEP_D, respectively, they are not in the skyline. More formally, a ***service skyline or S-Sky*** can be regarded as a set of SEPs that are not dominated by others in terms of all user interested QoWS aspects, such as response time, fee, and reputation. The service providers in the skyline represent the best tradeoffs among different user interested quality aspects [29]. As another example, SEP_F and SEP_G constitute the TripPlanner service skyline as highlighted in Table 1.

Example 3 (Composite service skyline). Let us consider the composition (referred to as TravelAssistance) of two services: TripPlanner and Map. A composite service execution plan or *CSEP* that consists of four operations (TripSearch, GetTrip, Geocode, GetMap) can be generated to access the composite service. The QoWS of a CSEP is computed by aggregating the QoWS of the SEPs in a similar way described in Example 2. A ***composite service skyline or*** **C-Sky** is a set of CSEPs that are not dominated by any other CSEPs. A naive way to find the C-Sky of the TravelAssistance composition is to generate all 20 possible CSEPs and then check service dominance among them.

TABLE 2
C-Sky Example

CSEP	Latency	Fee	Reputation
CSEP(A,F)	5.5	1.6	2
CSEP(A,G)	4.5	1.8	2
CSEP(D,F)	5.3	1.9	2
CSEP(D,G)	4.3	2.1	2

1.2 Computing the C-Sky

For a service composition with m services and assuming that there are $n_1, \ldots, n_m$ providers for each of the m services, $\prod_{i=1}^{m} n_i$ number of compositions need to be considered in order to find the C-Sky. For a complex composition with relatively large m (e.g., 10), a moderate number of providers for each service (e.g., 100) will introduce prohibitive overhead (e.g., 100^{10} compositions need to be considered). An intuitive solution that can significantly reduce the computational overhead is to compute the C-Sky from individual service skylines. This solution is based upon a key observation: a C-Sky can be completely determined by only considering the SEPs from S-Skies. Hence, only $\mathcal{N} = \prod_{i=1}^{m} k_i$ maximum number of compositions need to be evaluated, where k_i is the size of the ith S-Sky. Since the size of a S-Sky is typically much smaller than the number of providers, the computational cost can be reduced with several orders of magnitude. An example is given in Table 2. Instead of considering all 20 CSEPs, the C-Sky can be determined by only four candidate CSEPs, which are formed by combining the map and TripPlanner service skylines.

Based on the above observation, we developed two preliminary algorithms in our recent work for computing the composite service skylines [33]. The first algorithm, which is referred to as One Pass Algorithm or OPA, performs a single pass on the $\mathcal{N}$ compositions and outputs the skyline. The second algorithm, which is referred to as Dual Progressive Algorithm or DPA, progressively reports the skyline. Nevertheless, both algorithms fail to generate the C-Sky for moderate sized compositions (e.g., $m = 5$) within reasonable time. Hence, *a more efficient and scalable algorithm needs to be developed in order to cope with relatively complex compositions.*

We propose a novel Bottom-Up computation framework that enables to compute a C-Sky in a much more efficient fashion. We conduct an in-depth analysis of the preliminary service skyline algorithms with a focus on DPA. Although directly using DPA to compute a C-Sky is computationally expensive, DPA offers some key properties that inspire the design of a Bottom-Up Algorithm (BUA). BUA integrates a linear composition strategy into the proposed bottom-up framework to significantly boost the scalability of the algorithm, which makes it suitable to compute skylines for very complex service compositions. The analytical study shows that BUA is able to compute the C-Sky with a nearly optimal time complexity. We conduct

TABLE 3
Aggregation Functions

QoWS parameter	Aggregation function
$latency$	$\sum_{i=1}^{n} latency(op_i)$
$reliability$	$\sum_{i=1}^{n} \log(reliability(op_i))$
$availability$	$\sum_{i=1}^{n} \log(availability(op_i))$
fee	$\sum_{i=1}^{n} fee(op_i)$
$reputation$	$\sum_{i=1}^{n} reputation(op_i)$

an extensive set of experiments to evaluate the performance of the C-Sky algorithms.

2 Notations and Problem Definition

We describe the key notations used in this paper in Section 2.1. We then formally define the C-Sky computation problem in Section 2.2.

2.1 Notations

For clarity, we use the term *SEP* to specifically refer to the service execution plan for a single service. The *S-Sky* is used to refer to the skyline computed from the SEPs. We use the term *CSEP* to refer to the service execution plan for a composite service. Another important concept used in our approach is the *scores* of SEPs and CSEPs. A score is defined based on the QoWS of a SEP (or a CSEP). Recall that the QoWS of a SEP (or a CSEP) is computed by aggregating those of its member service operations. We focus on the type aggregation functions that can express the QoWS of a SEP (or a CSEP) as the sum of its operations' QoWS. 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 3. For an average function (e.g., aggregation of reputation), we use its aggregate version in our algorithms. The actual reputation can be easily derived from its aggregate version (by dividing the number of operations) after the C-Sky in computed.

Definition 1 (Score). *The score of a SEP is computed as* $\sum_j^k \sum_i^n q_j(op_i)$ *where* n *is the number of operations in the SEP,* q_j *is a QoWS attribute given in Table 3 and* k *is the number of QoWS attributes. The score of a CSEP is the sum of the scores of its SEPs.*

For example, we have score(SEP_A) $= 1.5 + 0.8 + 4 = 6.3$ whereas score(CSEP(A,F)) $=$ score(SEP_A) $+$ score(SEP_F) $= 6.3 + 8.8 = 15.1$. Based on the score definition and assuming that lower quality values are always preferred,[1] we have the following observation.

Property 1. *If* $SEP_1 \prec SEP_2$ *(resp.* $CSEP_1 \prec CSEP_2$*), then* $score(SEP_1) < score(SEP_2)$ *(resp.* $score(CSEP_1) < score(CSEP_2)$*).*

Proof. The score definition implies that the better the quality the less the score. $SEP_1 \prec SEP_2$ means that SEP_1 is as good as SEP_2 in all QoWS aspects and better than SEP_2 in at least one QoWS aspect, hence score(SEP_1) $<$ score(SEP_2). □

1. For some attributes, such as availability and reliability, where higher values signify better quality (referred to as positive attributes), we employ a simple strategy to convert them into negative attributes. More specifically, if q_j is a positive attribute, we set $q_j = q_j^{max} - q_j$, where q_j^{max} is computed by considering all providers for a service.

2.2 Problem Definition

We formally define the composite service skyline problem in this section. We then present a key observation that helps significantly reduce the searching space in finding the skyline.

Problem Definition. Given m services with d user interested QoWS attributes, where the ith service has n_i different providers, the problem of *composite skyline computation* is to compute the composite service skyline C-Sky over all possible service compositions.

It is worth to note that we do not consider candidate compositions with different number of services. In another word, the C-Sky is computed only from the composite services that compose the fixed m services.

The problem can also be formulated in a more generic sense as computing an *Aggregate Skyline* (or $\mathcal{AS}$) over m source tables $T_1, \ldots, T_m$, where each source table T_i has a set of columns C_i. Any two source tables T_i and T_j share a common set of columns, i.e., $(C_i \cap C_j) = \{c_1, \ldots, c_d\}$. For example, a travel agency database contains three tables that are used to store flight, hotel, and rental car information. Among other columns, all tables have three common columns, fee, service class, and user rating. When creating travel packages that include flight, hotel, and rental car, an aggregate table A is formed by aggregating the three source tables, where $A(i).c_j = f_j(T_1(i_1).cj, ..., T_m(i_m).cj), \forall j \in [1, d]$. $A(i)$ is the ith row of table A and c_j is an *aggregate column* computed by aggregation function f_j as discussed early in this section. More specifically, A is formed by performing a Cartesian product over the m source tables and aggregate columns of A are obtained by combining the matching columns in the source tables. An aggregate skyline consists of the rows in A, which are not dominated by any other rows on aggregate columns $c_i, ..., c_d$, i.e., $A(i) \in \mathcal{AS}$ if $\nexists A(j), A(j) \prec_{\{c_1,...,c_d\}} A(i)$. For the travel agency example, computing such an aggregate skyline is instrumental to locate the most attractive travel packages for its customers.

A straightforward way to compute the C-Sky is to first materialize all possible compositions (or all rows in the aggregate table) and then apply the existing skyline algorithm to find the skyline. However, as discussed in the Introduction section, computing the C-Sky in such a brute force manner is computationally intensive. The following observation helps significantly improve the performance.

Lemma 1 (Local search strategy). *Given* m *services* $S_1, \ldots, S_m$ *and the set of S-Skies* $SK_1, \ldots, SK_m$*, computed for each of them, the C-Sky over* $S_1, \ldots, S_m$ *can be completely decided by* $SK_1, \ldots, SK_m$*.*

Proof. For a CSEP, $\psi \in$ C-Sky, assume that it consists of m *SEPs*, $\text{SEP}_1, \ldots, \text{SEP}_m$, one from each service. Assume that each SEP is from the corresponding S-Sky, respectively, except for SEP_j, such that $\text{SEP}_j \notin SK_j$. Thus, there must be a $\text{SEP}'_j \in \text{SK}_j$ such that $\text{SEP}'_j \prec \text{SEP}_j$. Therefore, we can find a CSEP ψ' by replacing SEP_j in ψ with SEP'_j such that $\psi' \prec \psi$, which contradicts the fact that ψ is a skyline CSEP. □

Lemma 1 enables us to compute the S-Sky for each service (i.e., perform a local search) and then compute the C-Sky by only considering the SEPs in the S-Skies. This lemma directly leads to the development of the *One Pass Algorithm*, which performs a single pass on the CSEP space with a size of $\mathcal{N} = \prod_{i=1}^{m} k_i$ to compute the C-Sky. The *Dual Progressive Algorithm* leverages an expansion lattice and a heap to progressively compute the skyline. As shown in both the analytical and experimental studies, the high computational complexity of DPA makes it impractical to compute the C-Sky with a large number of services. Nevertheless, DPA offers a nice theoretical underpinning to a much faster *Bottom Up Algorithm*. BUA is built up a powerful bottom-up computational framework that exploits a linear composition strategy to achieve significantly better scalability and a nearly optimal time complexity.

3 Related Work

In this section, we give an overview of the existing works that are most relevant to the proposed service skyline algorithms.

Skyline computation has been intensively investigated in the database community [7], [28], [17], [24], [12], [14]. Block Nested Loops (BNL) and divide-and-conquer are among the first attempts to tackle the skyline computation problem [7]. BNL was extended by the Sort Filter Skyline (SFS) algorithm [12], which adopts a presorting scheme to improve the efficiency. SFS was further improved by the Linear Elimination Sort for Skyline (LESS) [14]. LESS exploits a small set of best data objects, referred to as an Elimination Filtering window (or EF window), to prune other objects in the initial pass of the external sorting. A special function is adopted in [4] that sorts the data points based on their minimum coordinate value, which avoids the scanning of the entire data set. The sorting-based algorithms can be used in conjunction with the local search strategy presented in Lemma 1 to compute composite service skylines. More specifically, we compute the S-Sky for each individual service and then instantiate the CSEPs from the S-Skies. The CSEPs are then sorted to generate the C-Sky. Our extensive experiments demonstrate that the proposed algorithms are computationally much more efficient than the sorting-based algorithms on computing composite service skylines.

Index structures, such as B-tree [7], have also been leveraged to improve the performance of skyline analysis. Two index structures were presented in [28] 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 structure [17], [24]. Since the composite services are generated dynamically, it is infeasible to precompute any index structures, which hinders us in exploiting the index-based approaches to compute the composite service skylines.

As dimensionality increases, the size of skyline may become very large and easily overload the end users. There are several key extensions on skyline analysis aiming to make it more flexible and adapt to user's preferences. A novel fuzzy skyline concept is proposed in [15], where five lines of extensions are presented to "fuzzify" a skyline query to increase its flexibility and discrimination power. k-dominant skyline relaxes the idea of dominance to k-dominance. A point p is said to k-dominate another point q if there are $k(\leq d)$ dimensions in which p is better than or equal to q and is better in at least one of these k dimensions [9]. A novel concept, called α-dominant skyline is proposed in [5] based on fuzzy dominance. An α-dominant skyline gives preference to services with a good compromise between QoS attributes. It also gives users the flexibility to control the size of the skyline. The fuzzy dominance, which signifies the degree to which p dominates q, leads to the definition of fuzzy dominating score [6]. The score enables to rank-order candidate points and returns the top-k candidates to a user.

Skyline computation has also been extended to a distributed environment, where data points are stored, accessed, and processed in a distributed fashion [3]. A progressive distributed skyline algorithm was proposed in [19] that can progressively report the skyline points in a distributed environment. Constrained skyline queries are investigated in a large-scale unstructured distributed environment [11]. A partition algorithm is exploited to divide distributed data sites into incomparable groups, which allows efficient parallel skyline query processing. A novel feedback-driven mechanism is developed in [37] to minimize the network bandwidth when computing a skyline on a data set that is horizontally partitioned onto geographically distant servers. Efficient skyline analysis techniques have also been developed in a Peer-to-peer (P2P) computing environment [13].

Jin et al. investigated the skyline operator on multi-relational databases [16]. The focus is on integrating efficient join methods into skyline computation based on the Primary Key and Foreign Key (PK-FK) relationship. Sun et al. studied a similar problem in the distributed environment [27]. Similar to [16], it also relies on the join attributes to prune candidate join tuples. The proposed C-Sky algorithms assume that Cartesian product is performed over multiple source tables. Hence, no PK-FK relationship or join attributes can be leveraged to prune the searching space. Cartesian product typically results in a much larger candidate space, which makes the problem more challenging. Furthermore, both [16] and [27] assume a standard join operation, which does not generate any aggregate columns. In contract, our algorithms essentially compute an skyline over aggregate columns obtained by combining the corresponding columns in the source tables.

In [29], techniques for generating "competitive products" are investigated. A product is formed by combining tuples from multiple source tables. The attributes of the products may be just a simple attribute copied from the source table or a merging attribute that is a weighted sum of multiple attributes from source tables. The objective is to find products that are not dominated by the existing products in the market and other newly generated products. The key pruning strategy is to first compute the skyline for each source table and then use these skylines to generate the final result. A major limitation is that if a product has more than one merging attributes, a brute force postprocessing is required, which is very computationally expensive. This

limitation may hinder the proposed algorithms from many applications because it is quite typical to have more than one merging attributes in a product or a service package, such as time, price, and so on. The pruning techniques proposed in our paper do not have such a limitation because we assume that each attribute of the CSEP is an aggregate from its SEPs. Therefore, the proposed algorithms can be applied to a much broader range of applications to efficiently compute the skylines.

Considering the expected large number of services competing to offer similar functionalities, quality aware service composition has received considerable attention nowadays [35], [31]. Most existing approaches formalize this as an optimization problem, which aims to find the best services to a composition with the best overall quality. Linear programming [35] and heuristic algorithms [31] have been exploited to tackle quality aware composition. A distributed local search strategy is integrated with mixed integer programming to further improve the optimization performance [1].

Skyline or similar concepts have been applied in the area of service computing. A service discovery framework was developed in [26] 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. The work that is most similar to ours is the one proposed [2], where skylines are used to select services for composition, reducing the number of candidate services to be considered. A strategy that is similar to Lemma 1 is used to improve the efficiency of skyline computation. A set of other strategies have also been proposed in [2] to select a set of representative skyline services. The algorithms proposed in our paper, on the other hand, focus on efficiently computing the entire composite service skyline. Since Lemma 1 directly leads to the development of the OPA algorithm, a comparison with OPA in our experimental study helps demonstrate the performance advantage of BUA over the skyline algorithm proposed in [2].

4 One Pass Algorithm

We present the OPA algorithm in this section. During the single pass of the CSEP space, OPA enumerates the candidate CSEPs one by one and only stores the potential skyline CSEPs. It outputs the skyline after all the candidate CSEPs have been examined. OPA requires that all the S-Skies are sorted according to the scores of the SEPs. OPA works as follows (shown in Algorithm 1). It starts by evaluating the first CSEP (referred to as CSEP_1) that is formed by combining the top SEPs from each S-Sky. It is guaranteed that $\mathrm{CSEP}_1 \in$ C-Sky because CSEP_1 has the minimum score so that no other CSEPs can dominate it. With the minimum score, CSEP_1 is expected to have a very good pruning capacity. Thus, OPA puts CSEP_1 on the top of the C-Sky so that the nonskyline CSEPs which are dominated by CSEP_1 can be pruned at the earliest time. After this, OPA continues to enumerate all other CSEPs one by one. CSEP_i will be inserted into the C-Sky if it is not dominated by any CSEP in C-Sky. Otherwise, the algorithm prunes CSEP_i and starts to check CSEP_{i+1}. During the checking process, whenever $\mathrm{CSEP}_j \in$ C-Sky is dominated by CSEP_i, C-Sky is updated by removing CSEP_j.

Algorithm 1. One Pass Algorithm
Input: m sorted S-Skies $SK_1, \ldots, SK_m$
Output: The C-Sky

```
1: N = ∏_{i=1}^{m} |SK_i|; // number of candidate CSEPs
2: CSEP_1 = Aggregate(SEP_11, ..., SEP_m1);
3: C-Sky.add(CSEP_1);
4: for all i ∈ [2, N] do
5:     CSEP_i = EnumerateNext(SK_1, ..., SK_m);
6:     IsDominated = False;
7:     for all j ∈ [1, |C − Sky|] do
8:         CSEP_j = C-Sky.get(j);
9:         if CSEP_i.score < CSEP_j.score then
10:            if CSEP_i ≺ CSEP_j then
11:                C-Sky.remove(j);
12:            end if
13:        else
14:            if CSEP_j ≺ CSEP_i then
15:                IsDominated = True;
16:                break;
17:            end if
18:        end if
19:    end for
20:    if IsDominated == False then
21:        C-Sky.add(CSEP_i);
22:    end if
23: end for
```

One outstanding issue with OPA is the *false positive* skyline CSEPs, which incur additional space and CPU cost. The false positives are generated because OPA has no restriction on the enumeration order of the CSEPs. Some early discovered CSEP that has been inserted into the skyline may be dominated by other later discovered CSEPs. These false positive CSEPs will stay in the skyline (which introduces space cost) and be compared with all the CSEPs 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 S-Skies (each of them is sorted on the scores of its SEPs). `EnumerateNext` first returns CSEP_1. It then keeps increasing the index of the mth S-Sky to enumerate the remaining CSEPs. When the index hits the end of the mth S-Sky, the index of $(m-1)$th S-Sky will increase by 1 and the index of the mth skyline will reset to 0. This will propagate to all other S-Skies until all CSEPs are enumerated. Since all the S-Skies are sorted, this process

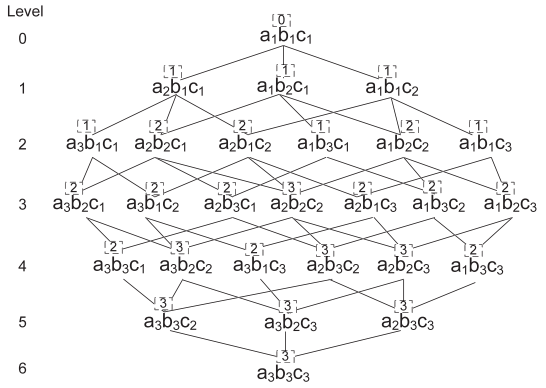


Fig. 1. Expansion lattice for $A(a_1, a_2, a_3)$, $B(b_1, b_2, b_3)$, and $C(c_1, c_2, c_3)$.

tends to enumerate the CSEPs by roughly following the ascending order of their scores.

5 Dual Progressive Algorithm

We present the DPA algorithm in this section. The underlying principle of DPA is the *dual progressive strategy*. By dual progressive, we mean that DPA *progressively* enumerates the CSEPs in an ascending order of their *scores* and also *progressively* reports the skyline CSEPs. Thus, it blocks on neither the input nor the output and is a completely pipelineable algorithm.

5.1 Basic Progressive Enumeration

Similarly to OPA, DPA also requires that the S-Skies are all sorted. The entire CSEP space can be enumerated systematically using a CSEP expansion lattice. Fig. 1 shows the expansion lattice for three S-Skies, $A(a_1, a_2, a_3)$, $B(b_1, b_2, b_3)$, and $C(c_1, c_2, c_3)$. The number of CSEPs generated from these skylines will be $|A| \times |B| \times |C| = 27$. Each node of the expansion lattice corresponds to a CSEP. In particular, the root node (referred to as r or n_1) corresponds to CSEP_1, i.e., the CSEP that is formed by the top SEPs from each S-Sky. $CSEP_1$ 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 S-Sky.

It is worth to note that the CSEP expansion lattice is completely different from the Low-Cardinality Lattice (LCL) used by Morse et al. [22] 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 CSEP 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 CSEP expansion lattice only defines the enumeration order between the nodes and makes sure that node n_i is enumerated after its ancestors. For nodes that do not have an ancestor-sucessor 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 $\mathcal{T}$ 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, which 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 CSEP_1 (i.e., n_1). Each enumeration step consists of two substeps: 1) `Extract`—the CSEP 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.

5.2 Parent Table

A major issue with the above basic implementation is that a single node could be generated from parent node expansion for multiple times, referred to as *node duplication*. Since a node could have up to m parents, where m is the number of S-Skies, the same child node will be generated when each of its parents is expanded. As shown in Fig. 1, 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 more than once, which results in a wrong skyline.

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 S-Skies, the parent table only uses up to $(\lfloor \log m \rfloor + 1)$ 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 S-Sky starts with 1. The number of parents for a given node n_i, represented as $(SEP_{1i}, \ldots, SEP_{mi})$, equals to the number of SEPs with an index greater than 1.*

Fig. 1 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 $\mathcal{P}$, 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 $\mathcal{H}$ is initialized with the root of the expansion lattice $\mathcal{T}$. Each enumeration step now consists of three substeps. `Extract` and `Expand` are the same as before except that `Expand` only generates the child nodes and does not insert any of them into $\mathcal{H}$. A new `UpdateCheck` step is added, which works as follows: For each newly generated child node, it

first updates $\mathcal{P}$ 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 $\mathcal{H}$. 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 S-Skies that form an expansion lattice $\mathcal{T}$
Output: The C-Sky
1: C-Sky $= \phi, \mathcal{H} = \phi$;
2: Initialize the parent table $\mathcal{P}$;
3: **while** $\mathcal{H} \neq \phi$ **do**
4: remove the top node n from $\mathcal{H}$;
5: **if** n is not dominated by any node in C-Sky **then**
6: C-Sky.add(n);
7: **end if**
8: $\mathcal{CN} = \text{expand}(\text{n}, \mathcal{T})$; // generate the child nodes
9: **for all** node $n_i \in \mathcal{CN}$ **do**
10: $\mathcal{P}(n_i) = \mathcal{P}(n_i) - 1$;
11: **if** $\mathcal{P}(n_i) == 0$ **then**
12: $\mathcal{H}$.add(n_i);
13: **end if**
14: **end for**
15: **end while**

Another key feature of the parent table is that it minimizes the heap size by completely avoiding the coexistence of a node and its ancestors in the heap (see Lemma 2 in Section 6). The heap size is a determining factor in the overall performance of the algorithm. Some other approaches can also be exploited to avoid node duplication. However, they cannot avoid the coexistence of a node and its ancestors hence increase the heap size. For example, we can use an array of flags to store the status of all the nodes. All the flags are initialized as 0. A flag is set to 1 after the corresponding node is inserted into $\mathcal{H}$. In this way, each node will only be inserted in the heap once. However, a node and its ancestors may coexist in the heap. For example, after the root node (a_1, b_1, c_1) is removed from $\mathcal{H}$, its three children $(a_2, b_1, c_1), (a_1, b_2, c_1)$, and (a_1, b_1, c_2) are inserted. Assume that (a_2, b_1, c_1) has the smallest score and is removed from $\mathcal{H}$. Now, $(a_3, b_1, c_1), (a_2, b_2, c_1)$, and (a_2, b_1, c_2) are inserted. Hence, (a_2, b_2, c_1) and (a_2, b_1, c_2) coexist in $\mathcal{H}$ with their parents (a_1, b_2, c_1) and (a_1, b_1, c_2). If we further assume that (a_3, b_1, c_1) is the next node removed from $\mathcal{H}$, this will result in the coexistence of (a_3, b_2, c_1) and (a_3, b_1, c_2) with their grandparents (a_1, b_2, c_1) and (a_1, b_1, c_2). In this regard, a lot of unnecessary comparisons between nodes and their ancestors will be conducted when they are inserted into the heap. The heap size will grow much faster and larger, which will slow down the overall performance of the algorithm. Another widely used duplicate avoidance strategy is the space partitioning method [20]. It assumes an order among different dimensions (e.g., A > B > C) and enumerates the candidate space based on this order. It completely avoids duplicate checking and does not need to store the status of the nodes. Nevertheless, it suffers the same issue as the flag array approach as nodes and their ancestors may coexist in the heap.

6 Bottom-Up Algorithm

We present the BUA algorithm in this section. We start by conducting an in-depth analysis of DPA, which demonstrates why DPA is computational expensive when the number of services is large. This leads to the development of the bottom-up framework and the linear composition strategy to improve the performance. The BUA algorithm is then presented, which is followed by a complexity analysis and a discussion.

6.1 An In-depth Analysis of DPA

We investigate the performance of DPA by examining the cost on each node in the expansion lattice. The time spent by DPA on node n_i consists of two major parts: heap operation (i.e., insertion and extraction) and skyline comparison. Assume that the size of the CSEP space is $\mathcal{N}$ and expected size of the CSEP skyline C-Sky is $\Theta((\ln^{d-1}\mathcal{N})/(d-1)!)^2$ [14]. The number of comparisons between skyline CSEPs is $|\text{C-Sky}|^2/2$, which is $o(\mathcal{N})$. Since the CSEP space is examined in a sorted order, the number of comparisons between skyline CSEPs and nonskyline CSEPs is bounded by $O(\mathcal{N})$ [14]. The cost of heap operation is determined by the size of the heap, which we examine in detail in the remaining part of this section.

Lemma 2. *A node and its ancestors (or descendants) cannot coexist in the heap.*

Proof Sketch. The parent table ensures that all the ancestors are removed from the heap before a node can be inserted. □

The order between the nodes that are incomparable (i.e., nodes that do not have a ancestor-successor relationship) with the expansion lattice is decided by the heap. Therefore, *the upper bound of the heap size is determined by the maximum number of incomparable nodes that can concurrently reside in the heap.* Some important properties of the expansion lattice help in providing an answer for this. For the ease of analysis, we assume that the index for each S-Sky starts from 0 and the size of each S-Sky is $(k+1)$. Thus, the index range for the jth S-Sky is $0 \leq i_j \leq k$. Assume that there are m S-Skies and the level number of the expansion lattice stats with 0.

With the above settings, we can immediately derive the number of nodes in each level of the expansion lattice. Based on how the expansion lattice is constructed (refer to Section 5.1 for details), the indices of the SEPs within a given node sum to the level number, upon which the node resides. Thus, for any node $(\text{SEP}_{i_1}, \text{SEP}_{i_2}, \ldots, \text{SEP}_{i_m})$ on level l, we have $i_1 + i_2 + \cdots + i_m = l$. As can be seen from Fig. 1, after subtracting 1 from each index of a given node (because the index here starts from 1 instead of 0), the sum of the indices equals to the corresponding level number.

2. The C-Sky size is just a rough estimate as the expected size is derived by assuming the independency of the attributes. However, QoWS attributes of CSEPs may not necessarily be independent.

Therefore, the number of nodes on a given level is actually the number of integral solutions of $i_1 + i_2 + \cdots + i_m = l, 0 \leq i_j \leq k$. This is given by the following equation:

$$N_l(m,k) = \sum_{j=0}^{[l/(k+1)]} (-1)^j \binom{m}{j} \binom{l+m-1-j(k+1)}{m-1}, \quad (1)$$

where $[l/(k+1)]$ is the integer part of $l/(k+1)$. Equation (1) has also been used in combinatorial analysis [25]. Another way to interpret this is that $N_l(m,k)$ is the coefficient of x^l in the expansion of $(1+x+\cdots+x^k)^m$. $N_l(m,k)$ achieves its maximum value at the middle level of the expansion lattice [8], i.e.,

$$\max(\{N_l(m,k) | 0 \leq l \leq mk\}) = N_{[\frac{mk}{2}]}(m,k). \quad (2)$$

The expansion lattice has a symmetrical structure, which can be justified by the following lemma.

Lemma 3. *The lth level and the $(mk-l)$th level of the expansion lattice have the same number of nodes, i.e., $N_l(m,k) = N_{mk-l}(m,k)$.*

Proof. The number of nodes on level l is determined by the number of integral solutions of $i_1 + i_2 + \cdots + i_m = l, 0 \leq i_j \leq k$. Suppose that $i_j = k - t_j, 1 \leq j \leq m$. Thus, we have

$$(k-t_1) + (k-t_2) + \cdots + (k-t_m) = l, 0 \leq t_j \leq k \quad (3)$$

$$\Rightarrow t_1 + t_2 + \cdots + t_m = mk - l, 0 \leq t_j \leq k. \quad (4)$$

Since there is a one-to-one correspondence relationship between i_j and t_j, the number of integral solutions of $i_1 + i_2 + \cdots + i_m = l, 0 \leq i_j \leq k$ equals to the number of integral solutions of $t_1 + t_2 + \cdots + t_m = mk - l, 0 \leq t_j \leq k$, which determines the number of nodes on level $mk - l$. □

Lemma 4. *The nodes on the same level of the expansion lattice are not comparable to each other.*

Proof. This is straightforward because the nodes on the same level do not hold an ancestor-successor relationship. □

Theorem 1. *The size of the heap $\mathcal{H}$ is bounded by the number of nodes on the middle level of the expansion lattice, i.e., $\max(|\mathcal{H}|) = N_{[\frac{mk}{2}]}(m,k)$.*

Proof. This actually means that the upper bound of the heap is achieved when all the nodes on the middle level of the expansion lattice concurrently reside in the heap. To prove this, we assume that the maximum size of the heap is achieved when there are nodes from some levels above the middle (same conclusion can be drawn for the levels below the middle due to symmetrical property stated in Lemma 3). Assume that n_i is from level i, where $i < [\frac{mk}{2}]$. All the nodes in the heap cannot be the successor of n_i. Suppose n_i is expanded at this point. Therefore, all its m child nodes can be inserted into the heap, with $m > 1$. This contradicts that the heap has already achieved its maximum size before n_i is expanded. □

We now asymptotically investigate the heap size based on (1). This provides an intuitive way to understand how heap size increases with some key parameters, including the size of the individual S-Skies (i.e., k) and the number of services (i.e., m). Specifically, let $l = pk$, where $0 \leq p \leq m$, and we have

$$\begin{aligned}\binom{l+m-1-j(k+1)}{m-1} &= \binom{(p-j)k-j+(m-1)}{m-1} \\ &= \frac{1}{(m-1)!} \times ((p-j)k-j+(m-1)) \\ &\quad \times ((p-j)k-j+(m-2)) \cdots \times ((p-j)k-j+(1)).\end{aligned}$$

On the right hand side, the product multiplies $(m-1)$ items in addition to $\frac{1}{(m-1)!}$, where each of these $(m-1)$ items contains $(p-j)k$. This allows us to rewrite the right hand side as the $(m-1)$ powers of k,

$$\binom{l+m-1-j(k+1)}{m-1} = \frac{(p-j)^{m-1}}{(m-1)!} k^{m-1} + \Theta(k^{m-2}). \quad (5)$$

The remaining terms can be expressed as $\Theta(k^{m-2})$ because the number of S-Skies (i.e., the number of services) is expected to be much less than the size of the S-Sky, i.e., $m \ll k$. We also have $j \leq [\frac{l}{k+1}] \leq m \ll k$. Therefore, the magnitude of the remaining terms is dominated by the highest power of k, which is $(m-2)$. Based on (5), (1) can be rewritten as

$$N_l(m,k) = \frac{\sum_{j=0}^{[l/(k+1)]} (-1)^j \binom{m}{j} (p-j)^{m-1}}{(m-1)!} k^{m-1} + \Theta(k^{m-2}). \quad (6)$$

Equation (6) shows that the heap size increases exponentially with the number of services although it may have a small constant factor. Also, the upper bound of the heap size cannot exceed k^{m-1}. This can be illustrated in a more intuitive manner. Assume that at certain point, the size of the heap exceeds k^{m-1}. In this case, we must have at least two nodes that are different from each other by only one SEP. Suppose that these two nodes are $(s_{1i}, \ldots, s_{jx}, \ldots, s_{mi})$ (referred to n_x) and $(s_{1i}, \ldots, s_{jy}, \ldots, s_{mi})$ (referred to as n_y), respectively, where $x \neq y$. Therefore, n_x is a ancestor (or a descendant) of n_y. The coexistence of n_x and n_y contradicts with Lemma 2.

Assume that the heap size is $|\mathcal{H}^{in}(n_i)|$ (or $|\mathcal{H}^{out}(n_i)|$) when node n_i is inserted into (or removed from) the heap. Then, the cost for inserting n_i into $\mathcal{H}$ is $\log(|\mathcal{H}^{in}(n_i)|)$. Although the cost for heap extraction is a constant, a reorganization of the heap is required after n_i is extracted, which has a cost of $\log(|\mathcal{H}^{out}(n_i)|)$. Since the total number of nodes in the expansion lattice is $\mathcal{N}$, the overall cost of heap operations is

$$\sum_{i=1}^{\mathcal{N}} [\log(|\mathcal{H}^{in}(n_i)|) + \log(|\mathcal{H}^{out}(n_i)|)]. \quad (7)$$

6.2 Bottom-Up Algorithm

As shown in last section, the performance of DPA is decided by two major factors: (F_1) heap operation and (F_2) skyline comparison. More specifically, F_2 is bounded by $O(\mathcal{N})$, where $\mathcal{N} = \prod_{i=1}^{m} k_i$, and $\mathcal{N}$ grows exponentially with the number of services. On the other hand, F_1 is determined by (7). Our complexity analysis shows that the upper bound

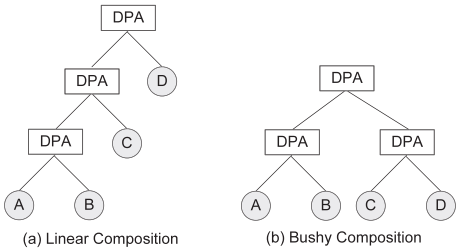


Fig. 2. Composition plans.

of the heap size also grows exponentially with the number of services due to Theorem 1. In this section, we present the BUA algorithm that is built up a novel bottom-up computation framework and exploits a linear composition strategy to gain significantly better efficiency and scalability. Furthermore, BUA also inherits all the nice properties of DPA, including progressive and pipelineable.

6.2.1 Bottom-Up Computational Framework

Theorem 2 lays out the theoretical grounding of the bottom-up computation framework.

Theorem 2 (Bottom-up strategy). *If CSEP $n_i^{(m+1)}$, represented as $(SEP_{1i}, \ldots, SEP_{(m+1)i})$ belongs to the $(m+1)-$C-Sky (i.e., the skyline computed over $(m+1)$ services), then CSEP $n_i^{(m)}$, represented as $(SEP_{1i}, \ldots, SEP_{mi})$, must belong to the m-C-Sky.*

Proof. Assume that $n_i^{(m)}$ is not in the m C-Sky. Thus, we can always find a $n_j^{(m)}$ such that $n_j^{(m)} \prec n_i^{(m)}$. By combining $n_j^{(m)}$ with $\text{SEP}_{(m+1)i}$, we have $(n_j^{(m)}, \text{SEP}_{(m+1)i}) \prec (n_i^{(m)}, \text{SEP}_{(m+1)i})$, which contradicts that $n_i^{(m+1)}$ is in the skyline. □

Theorem 2 forms the foundation of the bottom-up strategy. It implies that if $n_i^{(m)} \notin m-$C-Sky, then $n_i^{(m)}$ cannot be part of any CSEP in the $(m+1)-$C-Sky. Thus, we can safely prune $n_i^{(m)}$ when computing the m-C-Sky. This actually has the effect of pruning k CSEPs in the $(m+1)-$CSEP space, where k is the size of the $(m+1)$th S-Sky. *Instead of considering all the m services simultaneously (as DPA), the bottom-up algorithm progressively combines the m services together.* Specifically, BUA combines $m'(m' < m)$ SEP (or intermediate CSEP) skylines by using DPA in each step and keeps doing this until m is reached. Since DPA is completely pipelineable and the generated skyline CSEPs are automatically sorted, a skyline CSEP can be immediately used for the next-phase skyline computation. This guarantees the progressiveness of the bottom-up approach.

There are two remaining questions: (Q_1) *how to choose m'*, and (Q_2) *in what order to combine the temporary skylines.* The selection of m' is straightforward due to Theorem 2. To achieve the maximum pruning power of EP, we should always compute the skyline from the minimum number of SEP (or intermediate CSEP) skylines in each step, i.e., we choose $m' = 2$. The answer to Q_2 will lead to the concept of *linear composition plans*.

6.2.2 Linear Composition

For (Q_2), we first use an example to illustrate what it actually implies. We then present our solution for it. Fig. 2 shows two different ways to combine the SEP and intermediate C-Skies, which we refer to as *composition plans*. Fig. 2a gives a linear composition plan, where at least one child of a composition node is an S-Sky. On the other hand, a nonlinear or a bushy plan is shown in Fig. 2b. The following lemma helps us determine which type of plan to use.

Lemma 5 (Linear composition strategy). *The heap size used by DPA to compose two sets of skylines X and Y has an upper bound of $min(|X|, |Y|)$.*

Proof. For any two CSEPs in the heap, say n_s represented as (x_{is}, y_{js}) and n_t represented as (x_{it}, y_{jt}), $x_{is} \neq x_{it}$, and $y_{js} \neq y_{jt}$. Suppose $|X| < |Y|$. We assume there are $|X|+1$ CSEPs in the heap. Therefore, there must exist two CSEPs that share the same entry from X. This contradicts Lemma 2. □

Assume that the sizes of the S-Skies are all around k. Thus, the use of a linear composition plan in conjunction with the bottom-up computation guarantees that the maximum upper bound of the heap size is k. This is a significant improvement as compared with the upper bound of the heap size in DPA as shown in (6). Thus, the cost of any individual heap operation during the entire skyline computation is bounded by $\log(k)$. In contrast, in a bushy plan, the cost of the heap operations are determined by the sizes of the intermediate C-Skies. In addition, since the size of the intermediate C-Sky is typically larger than any of its children (we analyze this below), the cost of heap operations will keep increasing, which may become rather significant especially for a large number of services. On the other hand, the cost of heap operations for a linear plan is insensitive to the number of services, which helps it achieve a much better scalability.

The only remaining question now is to justify that the size of the intermediate C-Sky (referred to as $S_{(i,j)}$) is indeed larger than any of its two children (referred to as S_i and S_j, respectively). A straightforward way is to use the skyline cardinality estimation approach presented in [10], where we assume that the size of $S_{(i,j)}$ takes the form of $K_1 \log^{K_2}(|S_i| \times |S_j|)$ and parameters K_1 and K_2 can be estimated based on small data samples. Here, we present a more intuitive approach to roughly estimate the size of a intermediate C-Sky. Suppose that we want to compose two skylines A and B and summation is used as the aggregation function. The sizes of B and A are k and k', respectively. Skyline A consists of a_1, a_2, a_3, and a_4, as shown in Fig. 3a. For skyline B, instead of highlighting each individual skyline point, we use a curve to approximate the overall distribution of the skyline points for the ease of analysis. For the composition of A and B, we first compose each skyline point in A with B and then combine the result together. For example, for composing a_1 and B, we only need to shift the skyline curve of B with a distance of $a_1[x]$ along the x-axis and a distance of $a_1[y]$ along the y-axis. Fig. 3b shows the obtained result lists of composing a_1, a_2, and a_3 with B. Each result list is sorted based on the x values. We assume that there is a crossing point for every two consecutive compositions. For example, $a_1 + B$ and $a_2 + B$ cross at (x_1, y_1). (x_1, y_1) coincides with two *virtual points* $(b_{12}[x] + a_1[x], b_{12}[y] + a_1[y])$ and $(b_{21}[x] + a_2[x], b_{21}[y] + a_2[y])$,

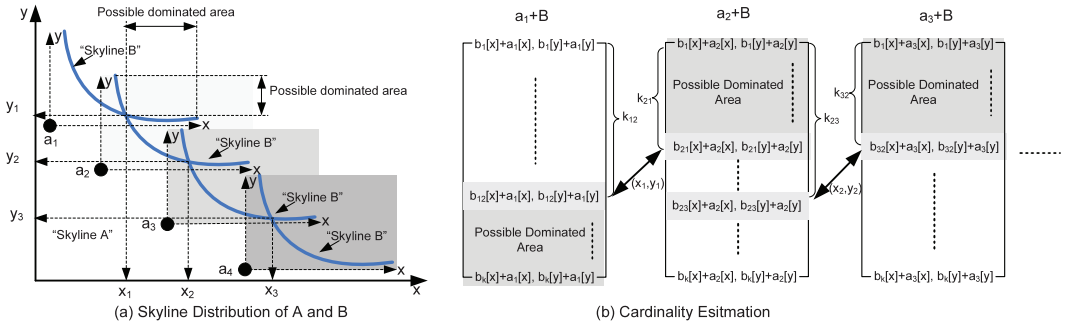


Fig. 3. Cardinality estimation for intermediate skylines.

that are from $a_1 + B$ and $a_2 + B$, respectively. We call these points virtual because there may be no actual points in the results list that corresponds to (x_1, y_1). Since $a_2[x] > a_1[x]$, we have $b_{12}[x] > b_{21}[x]$. Thus, we have $k_{12} > k_{21}$ (because the list is sorted on x). We generously assume that the points lay in the "possible dominated areas" (i.e., the lower part of $a_1 + B$ and the upper part of $a_2 + B$, as shown in Fig. 3a) are actually all dominated. Therefore, the remaining number of points is $(a_1, a_2, B) = k_{12} + (k - k_{21})$, which is greater than k because $k_{12} > k_{21}$. Similarly, we then combine $a_2 + B$ and $a_3 + B$ and have $(a_2, a_3, B) = (k_{23} - k_{21}) + (k - k_{32})$. We subtract k_{21} because these are dominated by $a_1 + B$. In addition, we keep the term $(k_{23} - k_{21})$ only if $k_{23} > k_{21}$, otherwise, $(a_2, a_3, B) = (k - k_{32})$. We keep applying this and obtain the final size of the skyline as $K_{AB} = \sum_{i=1}^{k'-1}(a_i, a_{i+1}, B)$. We have $K_{AB} > \max(k, k')$ because $(a_1, a_2, B) \geq (k + 1)$ and all the remaining $(k' - 2)$ items are greater than or equal to 1. We evaluate the effectiveness of linear composition in our experiments, which also verifies the above analysis. A formal proof of the above analysis may rely on the assumption of data distribution (e.g., assuming the independence of attributes like in [10]). We leave that as an interesting direction for future work.

6.3 Complexity Analysis

Similar to our previous analysis, we assume without loss of generality that there are m services and the sizes of the S-Skies are around k. The bottom-up approach employs $(m - 1)$ phases to compute the m-C-Sky. We use the notion $k_{(i+1)}$ to represent the size of the C-Sky generated from phase i. To adapt to this notion, we have $k_{(1)} = k$. For phase i, the complexity can be expressed as follows:

$$\underbrace{2 \times k \times k_{(i)} \times \log k}_{\text{(1) heap operation}} + \underbrace{k \times k_{(i)} + \frac{1}{2}k^2_{(i+1)}}_{\text{(2) skyline comparison}}, \tag{8}$$

where (1) is due to (7) and (2) is due to the analysis in the beginning of Section 6.1. Thus, we have the overall complexity as

$$\sum_{i=1}^{m-1} 2 \times k \times k_{(i)} \times \log k + k \times k_{(i)} + \frac{1}{2}k^2_{(i+1)} \tag{9}$$

$$= k(2 \log k + 1) \sum_{i=1}^{m-1} k_{(i)} + \frac{1}{2}\sum_{i=1}^{m-1} k^2_{(i+1)}. \tag{10}$$

Based on the above analysis on the intermediate skyline size, we have $k_{(i)} < k_{(i+1)} < k \times k_{(i)}$. Thus,

$$\frac{1}{k} < \frac{k_{(i)}}{k_{(i+1)}} < 1. \tag{11}$$

Assume

$$\max\left(\frac{k_{(i)}}{k_{(i+1)}}\right) = \frac{1}{p}, 1 \leq \forall i \leq m - 1 \text{and} 1 < p < k. \tag{12}$$

Therefore, we have

$$\sum_{i=1}^{m-1} k_{(i)} < \frac{1}{1 - \frac{1}{p}} \times k_{(i)} = \frac{p}{p-1} k_{(m-1)} \tag{13}$$

$$\sum_{i=1}^{m-1} k^2_{(i+1)} < \frac{1}{1 - \frac{1}{p^2}} \times k_{(i+1)} = \frac{p^2}{p^2 - 1} k^2_{(m)}. \tag{14}$$

Since we have $k_{(m)} > k_{(m-1)} > k$ and also note that $k_{(m)} = |\text{C} - \text{Sky}|$, the overall complexity can be derived as $O(\frac{p^2}{2(p^2-1)}|\text{C} - \text{Sky}|^2)$. This is nearly optimal because to compute a skyline with size $|\text{C} - \text{Sky}|$, at least $|\text{C} - \text{Sky}|^2/2$ skyline to skyline comparisons are required.

BUA also significantly reduces the space overhead of DPA. Due to Lemma 5, the maximum heap size is bounded by k. Meanwhile, the maximum size of the parent table is reduced to $k \times k_{(m-1)}$. This is significantly smaller than k^m, which is the size of the parent table used by DPA.

6.4 Discussion

An interesting question here is whether we can use OPA instead of DPA in BUA. We choose DPA over OPA due to two major reasons. First, by using DPA, BUA is able to progressively report the C-Sky. Second, OPA relies on the `EnumerateNext` function, which is most effective only when the skylines are sorted. However, since the skylines generated by OPA are no longer sorted, we may need to sort each intermediate skyline if using OPA with BUA, which will introduce significant overhead. Also, since a sorting needs to be performed before the intermediate skyline can be used for the next-phase computation, BUA is no longer pipelineable. On the other hand, if DPA is used, all the intermediate skylines are automatically sorted and can be directly used for the next-phase computation.

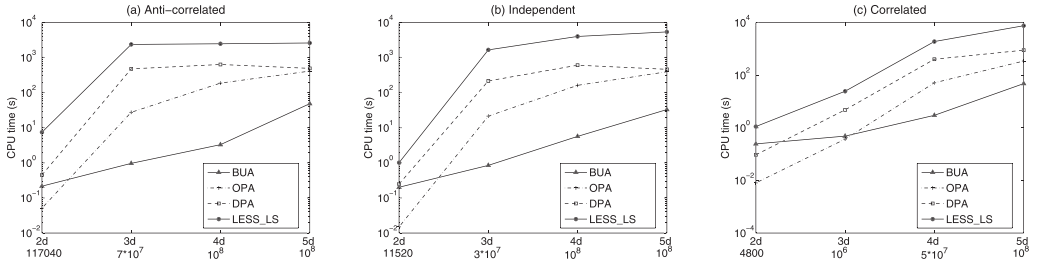


Fig. 4. CPU time versus d $(m = 4)$.

In addition, it is worth to note that DPA has a high computational complexity only when the number of services (i.e., m) is large, which is mainly due to the increase of the heap size. In BUA, we only use DPA to combine two skylines in each step. Due to Lemma 5, the heap size is bounded by k, where k is the size of the smaller skyline. In this case, DPA has a very similar performance with OPA, which has been demonstrated in the experiments.

7 Experimental Study

We use Java to implement all the proposed algorithms: OPA, DPA, and BUA. We run our experiments on a Mac Pro with two 2.8 GHz Quad-Core Intel Xeon processors and 4 G Ram under Mac OS X 10.6.7. 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 algorithms by using synthetic QoWS data. The QoWS attributes of service instances are generated in three different ways following the approach described in [7]: 1) *Independent QoWS* where all the QoWS attributes of service instances are uniformly distributed, 2) *Anticorrelated 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 use the BBS algorithm [24] to compute the S-Skies. Thus, the S-Skies are automatically sorted based on the scores of their SEPs.

We set the number of providers for each service (i.e., n) as 100,000, the number of QoWS attributes (i.e., d) in the range of 2-5, and the number of services (i.e., m) in the range of 2-10. When d and m change, the number of candidate compositions (i.e., $\mathcal{N}$) will change accordingly. $\mathcal{N}$ denotes the cardinality of the search space of the composite service skyline algorithms. In our experiments, the largest search space consists of 10^{20} candidate compositions.

In addition to the comparison among BUA, OPA, and DPA, we also implemented and compared our algorithms with a representative sorting-based algorithm, LESS [14]. We exploit LESS in two different ways to compute the C-Sky:

1. **LESS_LS**: LESS is integrated with the local search strategy as presented in Lemma 1. We first compute the S-Skies by using BBS. We then instantiate the CSEPs by aggregating the SEPs from the S-Skies and sort the CSEPs. LESS uses an EF window that holds 100 CSEPs to filter dominated CSEPs in the first pass of external sorting.
2. **LESS_BU**: LESS is integrated with the proposed bottom-up computation framework. In order to use LESS to compute a C-Sky in a bottom-up fashion, the entire computation process is divided into $(m - 1)$ steps (assuming that there are m services in the composition). Similar to BUA, in each step, LESS_BU combines one more S-Sky until all m S-Skies are combined. Since LESS is used, the intermediate CSEPs need to be sorted. Similarly, the EF window size in LESS is set to be 100.

As discussed in Section 6.4, the `EnumerateNext` function of OPA is most effective only when the skylines are sorted. As the skylines generated by OPA is no longer sorted, BUA is developed by integrating DPA into the bottom-up framework to achieve the best performance. To confirm this, we also implemented a bottom-up version of OPA, called **OPA_BU**, in which OPA is used to compute the C-Sky in a bottom-up fashion.

7.1 Effect of the Bottom-Up Strategy

We evaluate the effectiveness of the bottom-up strategy in this section. More specifically, we compare the CPU performance among OPA, DPA, LESS_LS, and BUA with different d and m. It is worth to note that the sizes of the S-Skies keep increasing with d, which is especially obvious with the anticorrelated QoWS. For instance, the size of a S-Sky typically exceeds 1,000 when $d \geq 5$. In practice, a service query may typically pose some other quality constraints, which help prune a large portion of the S-Sky. To avoid overly large S-Skies, we select the top 100 skyline SEPs (based on their scores) if the size of a S-Sky exceeds 100. Under this setting, the cardinality of the CSEP space (i.e., $\mathcal{N}$) may still become very large with the increase of the number of services. For example, $\mathcal{N}$ reaches the scale of 10^{20} when $m = 10$.

Fig. 4 compares the four algorithms, BUA, DPA, OPA, and LESS_LS with $m = 4$ and $d \in [2, 5]$. In the x-axis, we show the $\mathcal{N}$ values below the corresponding d values. Thus, the experiments also demonstrate how these algorithms scale with the cardinality of the CSEP space. $\mathcal{N} = 10^8$ for both $d = 4$ and $d = 5$ because the sizes of the S-Skies exceed 100 for large d and we select the top 100 skyline SEPs from these S-Skies. Since $m = 4$, we have $\mathcal{N} = 100^4 = 10^8$.

As can be seen, all the three proposed algorithms outperform the sorting-based algorithm LESS_LS. Among the three proposed algorithms, when $d = 2$, which also corresponds to a relatively small candidate CSEP space, OPA performs slightly more efficiently than BUA because it requires no overhead to maintain progressiveness. It also

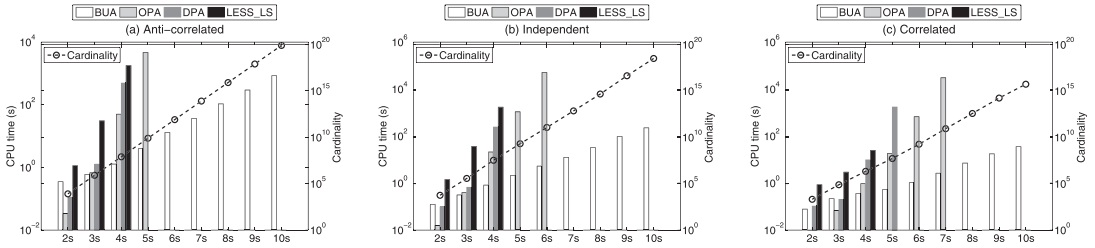


Fig. 5. CPU time versus m $(d = 3)$.

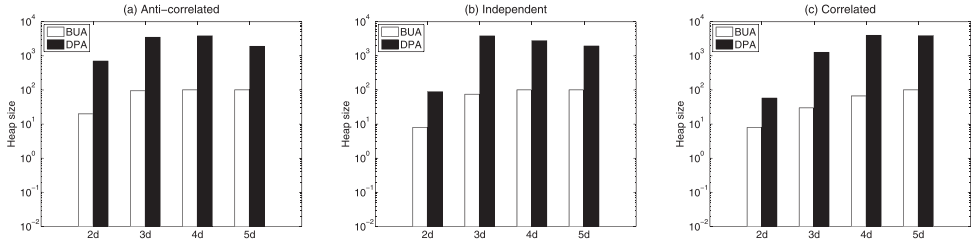


Fig. 6. Heap size versus d $(m = 4)$.

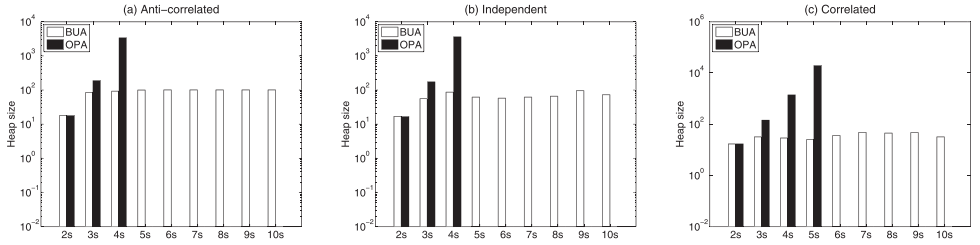


Fig. 7. Heap size versus m $(d = 3)$.

benefits from the effectiveness of the `EnumerateNext` function. BUA significantly outperforms other algorithms in all other cases. It is also interesting that the CPU cost of DPA does not necessarily increase with d. This is because the performance of DPA is decided by both the cardinality of the CSEP space $\mathcal{N}$ and the heap size $|\mathcal{H}|$. Since $|\mathcal{H}|$ may not necessarily increase with d (as shown in Fig. 6), the overall CPU cost of DPA does not necessarily increase with d.

Fig. 5 evaluates the impact of m. Similarly, we also show the $\mathcal{N}$ values. As can be seen, $\mathcal{N}$ reaches the scale of 10^{20} when m reaches 10. LESS_LS, DPA, and OPA all fail to output the skyline within reasonable time for moderate sized compositions (e.g., $m = 6$). In contrast, BUA scales very well with m and $\mathcal{N}$. It quickly (i.e., using less than 10 seconds) reports the skylines for moderate sized compositions (e.g., $m = 6$ for anticorrelated QoWS). It successfully computes the skyline for an extremely large CSEP space (i.e., in the scale of 10^{20}) within reasonable time.

The results also show that DPA is less efficient than OPA, which confirms our theoretical analysis about DPA. This is because the heap size $|\mathcal{H}|$ increases quickly with m. Nevertheless, when $m = 2$, the performance difference between OPA and DPA is almost negligible because the heap size remains small (bounded by the minimum size of the two S-Skies due to Lemma 5). This paves the way for the usage of DPA within the powerful bottom-up computational framework because m always equals to 2 in each intermediate computation step.

7.2 Effect of Linear Composition

We evaluate the effect of the proposed linear composition strategy by checking the maximum heap size during skyline computation. As shown in Fig. 6, the maximum heap sizes of BUA are significantly smaller than those of DPA with all $d \in [2, 5]$. Also, it is interesting to note that the maximum heap size does not necessarily increase with d for both DPA and BUA. For BUA, the maximum heap size is actually bounded by the maximum S-Sky size, which is due to Lemma 5. For DPA, it is determined by the maximum number of incomparable CSEPs that coincide in the heap (refer to Section 6.1 for details). Although with the increase of d, the chance of incomparability between CSEPs may increase as well, there are some other important factors that may affect the maximum heap size, such as the relative order of putting the CSEPs in the heap.

Fig. 7 evaluates the maximum heap size against m. The heap size of BUA is significantly less than that of DPA. The only exception is when $m = 2$, when BUA becomes identical to DPA. It is also important to note that the heap size of BUA is completely insensitive to m, which accounts for the good scalability of BUA. The maximum heap size of DPA, on the other hand, increases exponentially, which confirms our analytical result in Section 6.1. The DPA heap goes beyond the size of main memory for $m > 4$ $(m > 5)$ on anticorrelated and independent (correlated) QoWS.

7.3 Effect of Progressive Enumeration

We justify the effectiveness of progressive enumeration in the bottom-up computation framework by comparing the performance of BUA with OPA_BU and LESS_BU. Fig. 8

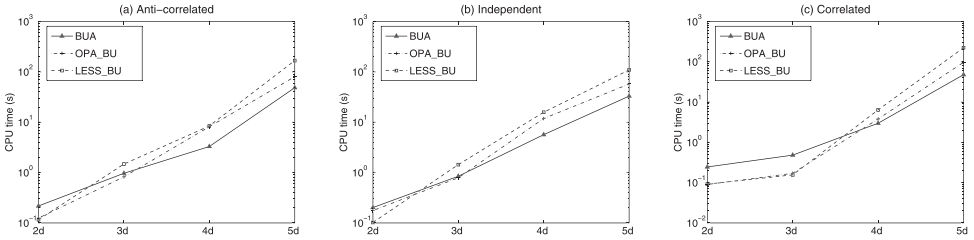


Fig. 8. CPU time comparison versus $d(m = 4)$.

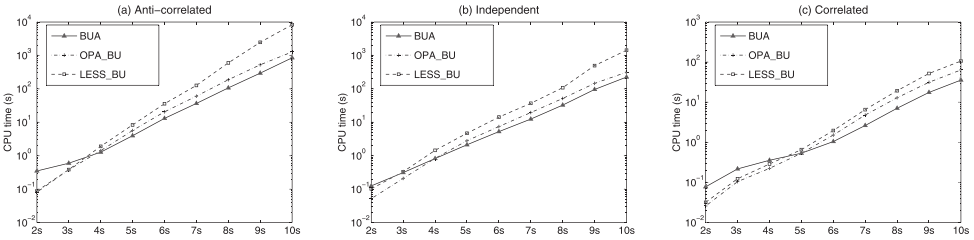


Fig. 9. CPU time comparison versus m $(d = 3)$.

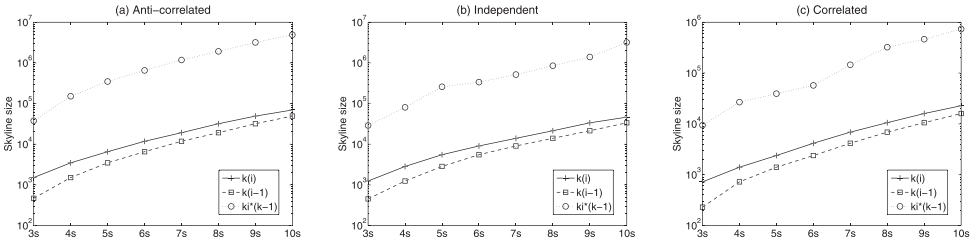


Fig. 10. $|\text{C-Sky}|$ versus $m(d = 3)$.

shows how CPU times of the three algorithms vary with d. Recall that a higher d value corresponds to a larger CSEP space. OPA_BU and LESS_BU perform slightly better than BUA when d is small (i.e., small CSEP space). Nevertheless, all three algorithms can generate the skyline very efficiently (i.e., using around or less than 1 second). As d increases, BUA becomes the most efficient algorithm, which further justifies its good scalability. Fig. 9 shows a very similar trend. These results confirm the effectiveness of integrating DPA with the bottom-up computation framework. It is also worth to note that both OPA_BU and LESS_BU exhibit significantly better performance and scalability than their non bottom-up versions. This further confirms the powerfulness of the proposed bottom-up computation framework.

7.4 Effect of the C-Sky Size

We evaluate the effect of the C-Sky size in this section. Fig. 10 shows that, in BUA, the C-Sky size for i services (referred to as $k_{(i)}$) is greater than skyline size for $(i-1)$ services (referred to as $k_{(i-1)}$) and less than $k_i \times k_{(i-1)}$ for all $m \in [3, 10]$. This is consistent with our theoretical analysis on the intermediate skyline size, which is summarized as (11).

8 Conclusion and Future Directions

We developed three skyline algorithms, OPA, DPA, and BUA, to select a set of "best" possible composite services. DPA employs an expansion lattice and a parent table to achieve progressiveness and pipelineability. BUA integrates a powerful bottom-up computation framework with a linear composition strategy that boosts the performance and scalability with orders of magnitude. We analytically and experimentally evaluated the algorithms and demonstrated that BUA is an efficient and scalable algorithm in computing composite service skylines.

Data incomparability has been considered as a key factor especially when computing skylines in a high-dimensional data space [36], [18]. An object-based space partition scheme is proposed in [36] that divides the data space into incompatible partitions to avoid unnecessary data comparisons. An interesting future direction is to exploit data incomparability to further improve the performance of the proposed composite service skyline algorithms.

References

[1] M. Alrifai and T. Risse, "Combining Global Optimization with Local Selection for Efficient Qos-Aware Service Composition," *Proc. 18th Int'l Conf. World Wide Web (WWW)*, 2009.

[2] M. Alrifai, D. Skoutas, and T. Risse, "Selecting Skyline Services for Qos-Based Web Service Composition," *Proc. 19th Int'l Conf. World Wide Web (WWW)*, 2010.

[3] W.-T. Balke, U. Guntzer, and J.X. Zheng, "Efficient Distributed Skylining for Web Information Systems," *Proc. Int'l Conf. Extending Database Technology (EDBT)*, 2004.

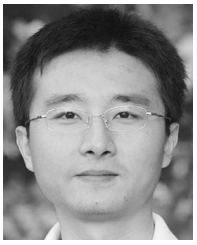
[4] I. Bartolini, P. Ciaccia, and M. Patella, "Efficient Sort-Based Skyline Evaluation," *ACM Trans. Database Systems*, vol. 33, no. 4, pp. 1-49, 2008.

[5] K. Benouaret, D. Benslimane, and A. HadjAli, "On the Use of Fuzzy Dominance for Computing Service Skyline Based on QoS," *Proc. IEEE Int'l Conf. Web Services (ICWS)*, 2011.

[6] K. Benouaret, D. Benslimane, A. HadjAli, and M. Barhamgi, "Top-k Web Service Compositions Using Fuzzy Dominance Relationship," *Proc. IEEE Int'l Conf. Services Computing (SCC)*, 2011.

[7] S. Borzsonyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *Proc. 17th Int'l Conf. Data Eng. (ICDE)*, 2001.

[8] D. Chakerian and D. Logothetti, "Cube Slices, Pictorial Triangles, and Probability," *Math. Magazine,* vol. 64, no. 4, pp. 219-241, 1991.
[9] C.-Y. Chan, H.V. Jagadish, K.-L. Tan, A.K.H. Tung, and Z. Zhang, "Finding K-Dominant Skylines in High Dimensional Space," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2006.
[10] S. Chaudhuri, N. Dalvi, and R. Kaushik, "Robust Cardinality and Cost Estimation for Skyline Operator," *Proc. 22 Int'l Conf. Data Eng. (ICDE),* 2006.
[11] L. Chen, B. Cui, and H. Lu, "Constrained Skyline Query Processing Against Distributed Data Sites," *IEEE Trans. Knowledge and Data Eng.,* vol. 23, pp. 204-217, Feb. 2011.
[12] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," *Proc. Int'l Conf. Data Eng. (ICDE),* 2003.
[13] B. Cui, L. Chen, L. Xu, H. Lu, G. Song, and Q. Xu, "Efficient Skyline Computation in Structured Peer-To-Peer Systems," *IEEE Trans. Knowlege and Data Eng.,* vol. 21, no. 7, pp. 1059-1072, July 2009.
[14] P. Godfrey, R. Shipley, and J. Gryz, "Maximal Vector Computation in Large Data Sets," *Proc. 31st Int'l Conf. Very Large Databases (VLDB),* 2005.
[15] A. Hadjali, O. Pivert, and H. Prade, "On Different Types of Fuzzy Skylines," *Proc. 19th Int'l Conf. Foundations of Intelligent Systems (ISMIS),* 2011.
[16] W. Jin, M. Ester, Z. Hu, and J. Han, "The Multi-Relational Skyline Operator," *Proc. IEEE 23rd Int'l Conf. Data Eng. (ICDE),* 2007.
[17] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," *Proc. Int'l Conf. Very Large Databases (VLDB),* 2002.
[18] J. Lee and S.-w. Hwang, "Bskytree: Scalable Skyline Computation Using a Balanced Pivot Selection," *Proc. 13th Int'l Conf. Extending Database Technology (EDBT),* 2010.
[19] E. Lo, K.Y. Yip, K.-I. Lin, and D.W. Cheung, "Progressive Skylining Over Web-Accessible Databases," *Data Knowledge Eng.,* vol. 57, no. 2, pp. 122-147, 2006.
[20] Y. Luo, X. Lin, and W. Wang, "Spark: Top-K Keyword Query in Relational Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2007.
[21] B. Medjahed and A. Bouguettaya, "A Multilevel Composability Model for Semantic Web Services," *IEEE Trans. Knowledge and Data Eng.,* vol. 17, no. 7, pp. 954-968, July 2005.
[22] M. Morse, J.M. Patel, and H.V. Jagadish, "Efficient Skyline Computation over Low-Cardinality Domains," *Proc. 33rd Int'l Conf. Very Large Databases (VLDB),* 2007.
[23] M. Ouzzani and B. Bouguettaya, "Efficient Access to Web Services," *IEEE Internet Computing,* vol. 37, no. 3, pp. 34-44, Mar. 2004.
[24] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An Optimal and Progressive Algorithm for Skyline Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2003.
[25] J. Riordan, *An Introduction to Combinatorial Analysis.* John Wiley and Sons, Inc., 1958.
[26] D. Skoutas, D. Sacharidis, A. Simitsis, and T. Sellis, "Ranking and Clustering Web Services Using Multicriteria Dominance Relationships," *IEEE Trans. Services Computing,* vol. 3, no. 3, pp. 163-177, July-Sept. 2010.
[27] D. Sun, S. Wu, J. Li, and A.K.H. Tung, "Skyline-Join in Distributed Databases," *Proc. ICDE Workshops,* pp. 176-181, 2008.
[28] K. Tan, P. Eng, and B. Ooi, "Efficient Progressive Skyline Computation," *Proc. 27th Int'l Conf. Very Large Databases (VLDB),* 2001.
[29] Q. Wan, R.C.-W. Wong, I.F. Ilyas, M.T. Ozsu, and Y. Peng, "Creating Competitive Products," *Proc. Int'l Conf. Very Large Databases (VLDB),* 2009.
[30] Z. Xu, P. Martin, W. Powley, and F. Zulkernine, "Reputation-Enhanced Qos-Based Web Services Discovery," *Proc. IEEE Int'l Conf. Web Services (ICWS),* pp. 249-256, 2007.
[31] Q. Yu and A. Bouguettaya, "Framework for Web Service Query Algebra and Optimization," *ACM Trans. Web,* vol. 2, no. 1, article 6, 2008.
[32] Q. Yu and A. Bouguettaya, "Computing Service Skyline from Uncertain QoWs," *IEEE Trans. Services Computing,* vol. 3, no. 1, pp. 16-29, Jan.-Mar. 2010.
[33] Q. Yu and A. Bouguettaya, "Computing Service Skylines Over Sets of Services," *Proc. IEEE Int'l Conf. Web Services (ICWS),* pp. 481-488, 2010.
[34] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient Algorithms for Web Services Selection with End-to-End Qos Constraints," *ACM Trans. Web,* vol. 1, no. 1, article 6, 2007.
[35] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Sheng, "Quality-Driven Web Service Composition," *Proc. 12th Int'l Conf. World Wide Web (WWW),* 2003.
[36] S. Zhang, N. Mamoulis, and D.W. Cheung, "Scalable Skyline Computation Using Object-Based Space Partitioning," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2009.
[37] L. Zhu, Y. Tao, and S. Zhou, "Distributed Skyline Retrieval with Low Bandwidth Consumption," *IEEE Trans. Knowledge and Data Eng.,* vol. 21, no. 3, pp. 384-400, Mar. 2009.



Qi Yu received the PhD degree in computer science from Virginia Polytechnic Institute and State University (Virginia Tech). He is an assistant professor in the College of Computing and Information Sciences at the Rochester Institute of Technology. His current research interests lie in the areas of service computing, databases, and data mining. His publications have mainly appeared in well-known journals (e.g., the *VLDB Journal*, the *ACM Transactions on the Web*, the *World Wide Web Journal*, and the *IEEE Transactions on Services Computing*) and conference proceedings (e.g., ICSOC and ICWS). He is a guest editor of the *IEEE Transactions on Services Computing*'s special issue on service query models and efficient selection. He frequently serves as a program committee member on service computing and database conferences (e.g, IEEE cloud, SOCA, CollaborateCom, IRI, ICSOC, and APSCC). He is also a reviewer for various journals (e.g., the *VLDB Journal*, the *ACM Transactions on the Web*, and the *IEEE Transactions on Service Computing*). He is a member of the IEEE.

Athman Bouguettaya received the PhD degree in computer science from the University of Colorado at Boulder in 1992. He is the head of the School of Computer Science and Information Technology at RMIT, Melbourne, Australia. He was a science leader at the CSIRO ICT Centre, Canberra, Australia. He was also previously a tenured faculty member in the Computer Science Department at the Virginia Polytechnic Institute and State University (commonly known as Virginia Tech). He currently holds adjunct professorships at the Australian National University, Canberra, the University of Queensland, Brisbane, Australia, and Macquarie University, Sydney, Australia. He is on the editorial boards of several journals, including the *VLDB Journal*, the *Distributed and Parallel Databases Journal*, the *International Journal of Cooperative Information Systems*, and the *IEEE Transactions on Services Computing*. He guest edited the *IEEE Internet Computing*'s special issue on database technology on the web and the *ACM Transactions on Internet*'s special issue on Semantic Web services. He served as the program chair of the 2008 International Conference on Service Oriented Computing (ICSOC 2008), the 20th Australasian Database Conference (ADC 2009), and the IEEE RIDE Workshop on web services for E-Commerce and E-Government (RIDE-WS-ECEG 2004). He has published more than 130 articles in journals and conferences in the area of databases and service computing (e.g., *IEEE Transactions on Knowledge and Data Engineering*, the *ACM Transactions on the Web*, the *International Journal on Very Large Data Bases*, SIGMOD, ICDE, VLDB, and EDBT). His current research interests include the foundations of web service management systems. He is a fellow of the IEEE.