Caching Hierarchical Skylines for Efficient Service Composition on Service Graphs

Hadeel Alhosaini
School of Computer Science
University of Technology Sydney
Sydney, Australia
Hadeel.Alhosaini@student.uts.edu.au

Xianzhi Wang
School of Computer Science
University of Technology Sydney
Sydney, Australia
Xianzhi.Wang@uts.edu.au

Lina Yao
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
Lina.Yao@unsw.edu.au

Yakun Chen
School of Computer Science
University of Technology Sydney
Sydney, Australia
Yakun.Chen@student.uts.edu.au

Guandong Xu
School of Computer Science
University of Technology Sydney
Sydney, Australia
Guandong.Xu@uts.edu.au

2022 IEEE International Conference on Services Computing (SCC) | 978-1-6654-8146-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SCC55611.2022.00015

***Abstract*—Service-oriented computing (SOC) is a paradigm for developing applications by reusing existing services. Through a standardized publishing, discovery, and composition process, SOC enables the orchestration of multiple (including third-party) services to constitute new applications. Hereby the quality of a composite service is fundamentally determined by its constituent services. To satisfy users' non-functional requirements, it is important to identify the optimal set of constituent services to participate in the composition. Practical applications usually require the optimal set to be identified with high efficiency and accuracy. This poses challenges to existing service composition methods as they either provide no accuracy guarantee or are inapplicable to large-scale problems. The challenges become more evident when considering service graphs, which contain multiple execution paths that could multiply the computational overhead. In this paper, we propose a hierarchical skyline-based approach for highly efficient service composition, which maintains and reuses varying levels of service skylines to accelerate service composition. We discuss how the skylines can be selectively computed, lazily updated, and efficiently retrieved for reuse. Experiments demonstrate the effectiveness of our approach.**

***Index Terms*—service composition, quality of service, hierarchical skyline, on-demand Updating**

I. Introduction

The prevalence of Service Oriented Architecture (SOA) and the increasing adoption of web service related technologies as the *de facto* standard have encouraged the exposure, discovery, and consumption of wrapped-functionalities of business applications for heterogeneous systems integration and inter-organizational interoperability [1]. As the ability of individual services is limited, the horizontal coupling of services via formal service process implementations (such as BPEL) is regarded as a promising means of satisfying complex application requirements [2]. The potential value-added (such as reduced engineering cost and fast customization ability) of such coarse-grained services, in turn, arouses the need for the efficient design and implementation of such services. The need becomes more urgent as the number of available services grows drastically on the Web. With the recent advances in cloud computing, Web of Things, and RESTful technologies, various applications can now be easily made available to the Web at low costs, with their functionalities exposed as APIs and consumable via simple URL strings. Until now, service composition has been a key research problem in various dynamic and configurable environments, such as cloud services, edge computing, and ad-hoc networks [3].

Generally, two approaches exist for service composition: the planning approach and the selection approach. The planning approach is a functional matching process that examines the functional compatibility between services to determine the feasible execution paths. The selection approach selects an appropriate set of services from categories of functionally-equivalent services to bind with a service process. In this study, we assume that a service process has been produced with different execution paths of it identified[1] based on a unified, consistent service ontology; so we focus on improving the efficiency and optimality of service composition. In particular, we aim at deciding on which execution path to select and which services to bind with the execution path to optimize the composite service's QoS. This problem is non-trivial for two reasons. First, service composition inherently suffers from low efficiency due to its geometrically expanding problem-scale. Second, it is more challenging to determine the optimal service composition solution on service graphs as the two issues, namely *the selection of the optimal execution path* and *the selection of the optimal constituent services*, are correlated. In particular, the former defines how the services are orchestrated and confines the scope of services that can be selected; the latter determines how well an execution path can meet the application requirements and makes it possible to differentiate the execution paths in terms of QoS.

[1]Since most service processes can be represented as graphs (typically the directed acyclic graphs or DAGs), we will hereafter call the service processes that contain multiple execution paths *service graph*s, for short.

In this paper, we propose to selectively precompute and cache hierarchical skylines to accelerate QoS-driven service composition over service graphs. In a nutshell, we make the following contributions:

- We propose a hierarchical skyline-based approach, which recomputes, maintains, and reuses varying levels of skylines to accelerate QoS-driven service composition over service graphs.
- We present the system architecture, models, and related algorithms to implement the approach. In particular, we employ dynamic programming and heuristic methods to cache only the most efficiency-effective skylines and use a lazy updating strategy to revalidate and update the cached skylines on-demand.
- We conduct comprehensive experiments to evaluate the proposed approach. Results show the approach significantly reduces the repetitive computation in the same domains at affordable costs and therefore greatly improves the efficiency of existing service composition methods.

The rest of the paper is organized as follows. Section II motivates our work with a running example. Section III introduces the architecture and components of our approach. As the further details, the methods for preparing skylines and making caching decisions are presented in Section IV, with the detection and updating methods of reusable skylines introduced in Section V. We report our experimental studies in Section VI, overview the related work in Section VII, and finally, give some concluding remarks in Section VIII.

II. A Motivating Example

We describe the motivation of our approach through a *product information retrieval* scenario in a mobile environment (based on [4]). In this scenario, users can retrieve the specification, reviews, or aggregated information of a product by providing a picture (e.g., captured by a phone) of the product's barcode and/or product-type. Table I lists the different types of functionalities that support this process, where each functionality corresponds to a group of functionally equivalent services with varied QoS. Fig. 1 shows the service graph formed by matching these functionalities. Note that, the functionalities with initial inputs (i.e., *JPG*) and final outputs (e.g., *Integrated Info.*) represent the entries and exits of the service graph, respectively. In particular, this example contains three exists.

Apparently, there are several execution paths for retrieving different types of product information in this service graph. The following shows some examples[2]:

(1) Sandy wants to retrieve reviews on a product through his phone. Two execution paths can individually serve this goal, i.e., $CR_1 \rightarrow PR_1 \rightarrow RR$ and $CR_2 \rightarrow PR_2 \rightarrow RR$.

(2) Bob wants to retrieve the specification of a product through his phone. Two execution paths can individually serve this goal, i.e., $CR_1 \rightarrow PR_1 \rightarrow SR$ and $CR_2 \rightarrow PR_2 \rightarrow SR$.

[2]For the sake of simplicity, we denote by $F_1 \rightarrow F_2$ the sequential execution of the two functionalities (i.e., F_1 precedes F_2) in a composite process.

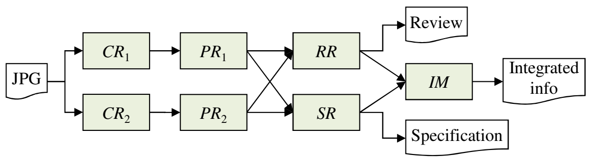


Fig. 1: The product information retrieval process

(3) Michael wants to retrieve aggregated information about the specification and reviews of a product. Here comes four execution paths that can individually serve this goal, i.e., $CR_1 \rightarrow PR_1 \rightarrow RR \rightarrow IM$, $CR_2 \rightarrow PR_2 \rightarrow RR \rightarrow IM$, $CR_1 \rightarrow PR_1 \rightarrow SR \rightarrow IM$, and $CR_2 \rightarrow PR_2 \rightarrow SR \rightarrow IM$.

(4) Johnny is a regular user who frequently makes the above three types of requests. His friends often do the same.

Each execution path in the above examples contains only some of the functionalities of the service graph, representing a subgraph. Given each subgraph, the optimal service set should be identified and evaluated for each execution path before the optimal execution path can be determined. Traditional service selection methods independently compute for each execution path, with little information shared between the different computations. This incurs multiplied computation load when compared with the computation regarding a single execution path. In fact, given a specific application domain, some functionalities are frequently requested by users and some users may raise similar or identical requests, like what Johnny and his friends do in example (4). This provides a clue of reusing the shared computation regarding different requests to accelerate the QoS optimization of service composition. Moreover, consider a system that has numerous active users. Requests regarding the same or similar service subgraphs may occur frequently. Such situation makes the shared computing even more meaningful.

III. Approach

Our approach aims at computing and reusing hierarchical skylines to accelerate the QoS optimization of service composition within the same application domain. While skyline services represent the Pareto-optimal subset of services, their superiority is independent of users' constraints and preference on QoS. This makes skylines[3] generic enough to serve requests with varying constraints and optimization objectives of service composition.

A. Architecture

Fig. 2 shows the system architecture of our approach, which comprises components in a three-phase service composition procedure. The first phase prepares the skyline services for fast retrieval and reuse in future compositions. Promising functionalities and subprocesses are identified from historical composition and indexed as nodes in a cache model, named *CSTR*, with their corresponding skylines precomputed and

[3]A *service skyline* (or *skyline* for short) is a set of *skyline services* on a specific task, where a *task* refers to either a functionality or a subprocess.

TABLE I: Specifications of the functionalities involved in the retrieval process

Functionality	Function Type	Input	Output
CR_1	Code recognition	Picture	Barcode
CR_2	Code recognition	Picture	Typecode
PR_1	Product ID retrieval	Barcode	Product ID
PR_2	Product ID retrieval	Typecode	Product ID
RR	Review retrieval	Product ID	Product review
SR	Specification retrieval	Product ID	Product specification
IM	Information aggregation	cReviews, Specification	cAggregated information

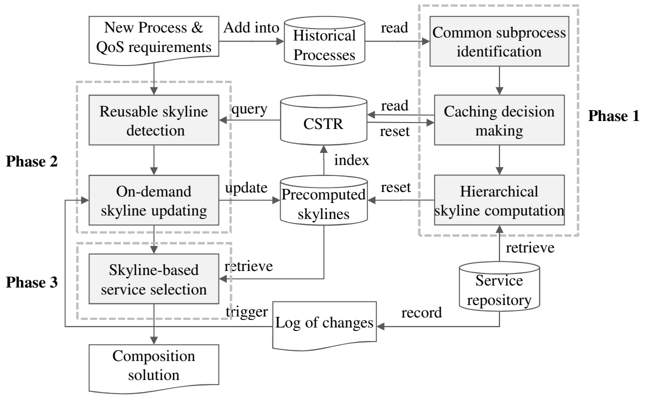


Fig. 2: Architecture for hierarchical skyline-based service composition

stored as records in the main memory. The *CSTR* is a DAG, which serves as a dictionary where each node has access to its corresponding skylines for reusable skyline detection and updating. The second phase deals with new requests regarding a specific service graph. The service graph is matched with *CSTR* to detect a set of reusable skylines. These skylines are then checked against the log of changes (which records the changes of available services of each functionality) in the service registry. Whenever a skyline is detected as non-optimal, it is updated according to the log to ensure accuracy. Finally, the third phase selects the optimal execution path and services to fulfill the composition, without having to recompute the detected skylines from scratch. The first phase is performed offline on a periodical basis while the other two phases are triggered at real-time by service composition requests.

B. Cache Model

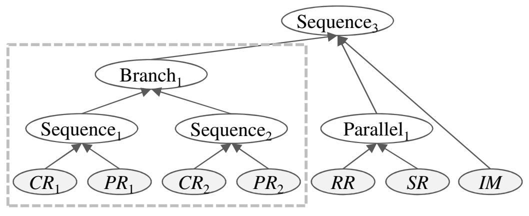
The precomputed skylines should be well-organized to facilitate future retrieval and reuse. In this approach, we establish a topological indexing structure called *CSTR*, where each node represents a functionality or subprocess and contains access to the corresponding skylines. In *CSTR*, a node pointing to another indicates the former is a functionality or subprocess of the latter. Suppose P_1 and P_2 are two subprocesses with their skylines, say SL_1 and SL_2, both cached. By referring to the *inclusion* relationship between processes, we define the *supporting* relationship between P_1 and P_2 as follows: If P_1 is a functionality or subprocess of P_2, we say SL_1 *support*s SL_2 or there exists a **supporting** relation from SL_1 to SL_2. Apparently, it is a transitive relationship—we can infer from "SL_1 supports SL_2" and "SL_2 supports SL_3" that "SL_1 supports SL_3". Since the subprocesses are of different granularities, the nodes and their relations in $CSTR$ form a layered structure, where lower-level skylines can be used to compute upper-level ones; so besides *id* and the *functionality/subprocess* it stands for, each node records the *last update time*, *usage frequency*, and *entry to the corresponding skylines*. As an example, Fig 3a shows the *CSTR* structure for indexing all levels of skylines of the product info. retrieval process (Fig. 1), where *CSTR* follows the same hierarchical structural patterns of the service process.

C. Algorithm

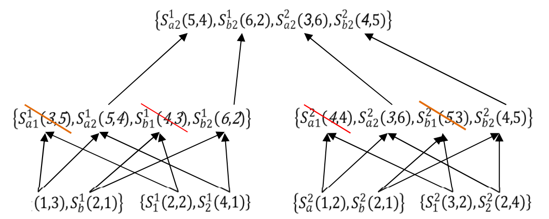
Traditional service composition methods only reuse skylines at the functionality level. In this paper, we propose to reuse varying levels of skylines (indexed by *CSTR*) to accelerate service composition. Algorithm 1 shows the procedure for hierarchical skyline-based service selection (HSBS). It first initializes three sets to hold the functionalities, logs, and reusable skylines, respectively, relevant to the service graph G. Given the service graph G, the reusable skylines (represented by the common functionalities and subprocesses of G and the indexing structure *CSTR*) are first detected (line 2). Since the log only records changes regarding single services, Algorithm 1 continues to find the log related to the detected skylines, i.e., all the log related to functionalities covered by the identified skylines (lines 3-8), and update these skylines layer by layer according to the log (lines 9), when necessary, to make them accurate. Then, service selection is performed regarding each execution path (denoted by p) of the service graph (lines 10-13), and finally, the optimal solution is the best solution among the optimal solutions of all execution paths (line 14).

In above description, Algorithm 1 employs three external algorithms to detect (i.e., RSDA) and update (i.e., ISUA) the reusable skylines, and to perform service selection on each execution path (i.e., ESSA). We introduce the first two algorithms in the following sections but omit the introduction of ESSA, as it can be easily fulfilled by existing QoS-based service selection methods.

Let $F(SL_i)$ be the set of functionalities covered by the i-th reusable skyline in MR, i.e., SL_i. By introducing hierarchical skylines, the number of tasks involved in service selection is reduced from the original number, say N, to $N - |\cup_{SL_i \in MR} F(SL_i)| + |MR|$, and the number of candidate services may



(a) A sample CSTR structure


$\{S^1_{a2}(5,4), S^1_{b2}(6,2), S^2_{a2}(3,6), S^2_{b2}(4,5)\}$
$\{S^1_{a1}(3,5), S^1_{a2}(5,4), S^1_{b1}(4,3), S^1_{b2}(6,2)\}$
$\{S^2_{a1}(4,4), S^2_{a2}(3,6), S^2_{b1}(5,3), S^2_{b2}(4,5)\}$
$(1,3), S^1_b(2,1)\}$
$\{S^1_1(2,2), S^1_2(4,1)\}$
$\{S^2_a(1,2), S^2_b(2,1)\}$
$\{S^2_1(3,2), S^2_2(2,4)\}$

(b) Sample skyline service reduction

Fig. 3: Indexing structure and skyline reduction for the motivating example

Algorithm 1 Hierarchical Skyline-Based Service Selection (HSBS)

```
Input: a service graph $G$, the indexing structure $CSTR$, and the log of changes $Log$.
Output: the optimal service composition solution.
1:  $FS \leftarrow \emptyset$, $Logset \leftarrow \emptyset$, $RS \leftarrow \emptyset$
2:  $MR \leftarrow$ RSDA($G$, $CSTR$)
3:  for $e \in MR$ do
4:      $FS \leftarrow FS \cup$ the functionalities covered by $e$ in $CSTR$
5:  end for
6:  for $t \in FS$ do
7:      $Logset \leftarrow Logset \cup$ records related to $t$ in $Log$
8:  end for
9:  ISUA($CSTR$, $Logset$)
10: for $p$ in $G$ do
11:     $RS_p \leftarrow$ ESSA($CSTR$, $p$)
12:     $RS \leftarrow RS \cup \{RS_p\}$
13: end for
14: return $(p^*, RS_{P^*})$, where $RS_{P^*}$ is the best of $\{RS_p\}_{p=1,2,\cdots,P_G}$ (suppose $P_G$ is the number of execution paths in $G$).
```

also decrease. As an example, Fig. 3b shows the bottom-up skyline computation process for the subprocess encircled by the dashed rectangle in Fig. 3a, where each functionality has originally two skyline services. Each $s(v_1, v_2)$ represents a service's quality values on two QoS attributes. For either of the two attributes, a higher value indicates better quality. By computing the hierarchical skylines, the total number of candidate services is reduced from 8 to 4. Since execution paths in the same service graph often overlap, given a skyline shared by different execution paths, even if it requires updating, it is updated only once but reused immediately by different computations regarding these execution paths. This indicates it is economical to cache and reuse the shared skylines.

IV. Skyline Preparation

In this section, we describe methods for identifying promising subprocesses from historical service graphs and set up criteria for determining which corresponding skylines to index and precompute for future reuse.

A. Identifying Common Subprocesses

Intuitively, the more frequent a subprocess occurs in historical service graphs, the more likely the corresponding skylines are to be reused in the future. Given a set of historical service graphs, we identify and organize the frequently occurring subprocesses in two steps: (1) construct a topological structure of all subprocesses and functionalities, i.e., *CSTR** by comparing and merging the historical service graphs and obtain their *usage frequency*, and (2) refine the structure to keep only the nodes that have the potential to be effective in improving the efficiency. The skylines corresponding to these nodes can then be precomputed, stored into records, and indexed for reuse by a caching decision-making process (Section IV-B).

In particular, step (1) constructs *CSTR** by repeatedly merging it with the historical service graphs, one at a time. The hierarchical structural patterns of service graphs (as shown in Fig. 3a) provide the rationale of identifying the common functionalities and subprocesses of *CSTR** and a service graph in a bottom-up manner to reduce repetitive computation. The time complexity of this step is $O(M^2)$, where M is the number of nodes in the final *CSTR**. Step (2) directly removes the non-recurring functionalities and subprocesses, which are apparently efficiency-ineffective, from *CSTR**. During this process, the nodes in *CSTR** are checked one by one with respect to their occurrence frequency in a top-down manner, where a node is removed from *CSTR** by the following rules:

- Both the top-layer (with zero outdegree) and the bottom-layer nodes (with zero indegree) are removed directly, with all their related edges also removed from *CSTR** .
- A middle-layer node is removed with all its direct children handed over to each of its parents, meaning all its children nodes turn to directly support each of its parent nodes.

The time complexity of step (2) is also $O(M^2)$.

B. Making Caching Decisions

Caching decisions are subject to both effectiveness and size constraints. The cached skylines should effectively reduce the computation load and meanwhile be kept at a reasonable size. Usage frequency, granularity, and costs of retrieving and updating together define the effectiveness of a skyline. Generally, skylines of finer-granularities are more likely to be reused and incur less cost, but their ability to reduce the number of tasks and candidate services is also limited.

To predicting the impact of caching a skyline is a challenging task as the impact of the cached skylines is usually interdependent. It becomes more complicated when varying levels of skylines and their maintenance costs are concerned. For this reason, the cached skylines are better measured as a group. We define the optimal caching strategy as the problem of maximizing the overall impact of computation load reduction under constraints on the total cache size, i.e.,

$$\begin{aligned} &\underset{i}{\text{maximize}}\ \mathcal{I}(\cup_{i=1}^{|CSTR^*|}\{t_i \cdot x_i\}) \\ &\text{s.t.} \sum_{i=1}^{|CSTR^*|} size(t_i)\cdot x_i < size_{max} \text{ and } x_i \in \{0,1\} \end{aligned} \tag{1}$$

where $|\mathit{CSTR}^*|$ is the number of nodes in *CSTR**, t_i is the i-th node in $|\mathit{CSTR}^*|$, which corresponds to the i-th skyline to be cached, $size_{max}$ is the maximum size of skylines, $\mathcal{I}(\cdot)$ is the function that evaluates the overall impact of caching a group of skylines, and x_i is an indicator, which equals 1 if the skyline of t_i is cached and 0 otherwise.

This problem is a variant of the 0/1 knapsack problem and can be solved directly using dynamic programming. The general time complexity of a dynamic programming solution is $O(|\mathit{CSTR}^*|\cdot size_{max}\cdot C(\mathcal{I}))$, where $C(\mathcal{I})$ is the complexity of calculating $\mathcal{I}(\cdot)$. Note that, in Eq. (1), we do not define constraints to ensure each cached skyline is efficiency-effective, as dynamic programming automatically attends to this issue. Due to the difficulty of defining an accurate evaluation function, $\mathcal{I}(\cdot)$, and due to the severe time complexity of dynamic programming, we define a heuristic method to bypass the above complication. In particular, we define the heuristics as follows:

$$h_i = |F(t_i)| \cdot f_{t_i}^{us} \tag{2}$$

where $F(t_i)$ is the set of functionalities covered by the node t_i and $f_{t_i}^{us}$ is the usage frequency of t_i in historical compositions. Note that, although both the reduction effect and the maintenance cost are related to $F(t_i)$, a larger $|F(t_i)|$ has a positive effect on the performance as long as $f_{t_i}^{us} > f_{t_i}^{ud}$, where $f_{t_i}^{ud}$ is the update frequency of the corresponding skyline of t_i. For this reason, we cache only the skylines that satisfy the above inequation. Since both factors in Eq. (2) positively affect the caching probability of a skyline, a high h_i value indicates a better skyline.

To implement the heuristic method, we simply add the skylines with higher h_i values in priority to the cache. Intuitively, a skyline is worth caching only if the benefits outweigh the costs. However, since h_i is a relative measure, it can only be used to prioritize the skylines without being able to predict whether it is worthwhile to cache a specific skyline. Although we can limit the skylines to cache by restricting their total size, the size we define may not always be optimal. In this paper, we address this issue empirically by periodically attempting to remove or add a skyline from/to the cache. If the performance improves after the removal (resp., addition) of a skyline, we continue trying removing (resp., adding) a new skyline from (resp., to) the cache; otherwise, the newly removed (resp., added) skyline would be recovered (resp., removed) and the next-round trial would be about adding (resp. removing) a skyline instead. The trials continue until neither adding or removing a new skyline could improve the performance of service composition.

After solving the optimization problem, every functionality or subprocess whose skyline is not to be cached is removed from *CSTR**. The remaining functionalities and subprocesses and their relations together form *CSTR*. We compute the skylines following the structural relations of their corresponding nodes in *CSTR*, i.e., higher-level skylines are computed from lower-level ones. This bottom-up computing procedure is analogical to that of dynamic programming. We will not introduce in detail the computing methods due to the limited space.

V. Skyline Reuse

Given a new request regarding a service graph G, our approach first matches G with *CSTR* to detect the reusable skylines, and then updates the detected skylines according to the log of changes since their last updates in the service registry.

A. Reusable Skyline detection

To detect the reusable skylines is a simpler case of *CSTR* construction (Section IV-A). The former matches a service graph with *CSTR* to detect their common subprocesses, so as to enable the reuse of their corresponding skylines. The latter iteratively merges the historical service graphs one by one into *CSTR* by adding new nodes and their relations to form a structured collection of all process patterns in a specific domain. Identifying the common subprocesses is the fundamental technique that enables this merge.

Algorithm 2 illustrates the procedure for reusable skyline detection It first initializes a set to hold the final results (line 1) and then identifies the common functionalities (lines 2-5), where $CommonSet$ contains the common functionalities between the two node sets and is gradually added to the result. Following that, the common subprocesses are detected iteratively in a bottom-up manner and added to the final result (lines 6-11).

The time complexity of Algorithm 2 is $O(V^2 \log V)$, where V is the bigger number of nodes in *CSTR* and in *G*. Specially, among the detected reusable skylines, only those of the largest granularities are reused. For example, given a sequential process $CR_1 \rightarrow PR_1 \rightarrow SR$ and the *CSTR* shown in Fig. 3a, only two skylines that correspond to $sequence(CR_1, PR_1)$ and SR would be reused, where $sequence(CR_1, PR_1)$ is a common subprocess and SR is a common functionality. In contrast, the skylines of CR_1 and CR_2 would not be reused as they are both covered by the skyline of $sequence(CR_1, PR_1)$.

B. Updating Reusable Skylines

We consider a dynamic environment where the available services may change over time. In particular, a service may be registered, deactivated or modified in terms of its declared

Algorithm 2 Detecting Reusable Skylines (RSDA)

```
Input: CSTR and a service graph G, both represented as DAGs.
Output: a set of common subprocess of CSTR and G.
 1: Initialize $CSP \leftarrow \emptyset$
 2: $Cset \leftarrow$ nodes with the indegree of zero in CSTR
 3: $Nset \leftarrow$ nodes with the indegree of zero in G
 4: $CommonSet \leftarrow Cset \cap Nset$
 5: $CSP \leftarrow CSP \cup CommonSet$
 6: while $CommonSet \neq \emptyset$ do
 7:     $Cset \leftarrow$ nodes supported only by element of commonSet in CSTR
 8:     $Nset \leftarrow$ nodes supported only by commonSet in NP
 9:     $CommonSet \leftarrow Cset \cap Nset$
10:     $CSP \leftarrow CSP \cup CommonSet$
11: end while
12: return $CSP$
```

QoS. The log of changes records such changes in terms of log records, where each log record describes the time, operation (i.e., register, deactivate, or modify), and the target service. When a change happens, the related skylines are revalidated against this log and updated if they turn inaccurate—this happens when some skyline service becomes invalid or new skyline services emerge.

We adopt a lazy updating strategy, which selectively and reactively updates the affected skylines and automatically adapts its behavior according to the frequency and distribution of requests—the more frequently a functionality/subprocess is used by recent requests, the more frequently its corresponding skyline is revalidated and updated (if necessary), and the higher possibility this skyline can be reused directly by future requests.

Algorithm 3 shows the skyline updating procedure (*ISUA*), which iteratively updates the affected skylines in a bottom-up manner, following the structural relations of their corresponding nodes in *CSTR*. Given the log records regarding functionalities related to the skylines to be used for a service graph, it updates a hierarchy of skylines affected by every record one by one. Regarding each record, it first updates the skyline for the relevant functionality and then update the higher-level skylines (i.e., nodes in $CSTR$) supported by the functionality layer by layer.

Suppose U is the complexity of updating each skyline, the time complexity of Algorithm 3 is $O(|Log|H \cdot U)$, where H is the length of the longest path in *CSTR*. The upper bound of the time complexity of updating a skyline equals the complexity of computing the skyline from scratch, i.e., $O(N^2R)$ for functionality-level skylines and $O(n^MR)$ for subprocess-level skylines, where N is the service number per functionality, R is the QoS dimensionality, M is the number of skylines or tasks from which the skyline is computed, and n is the number of (skyline) services per skyline or task.

Algorithm 3 Iterative Skyline Updating (ISUA)

```
Input: the indexing structure CSTR, and the log of changes related to the skylines to be reused, Log.
Output: updated skylines
1: for $r \in Log$ do
2:     $c \leftarrow$ the functionality described by $r$ in CSTR
3:     update $c$'s corresponding skyline
4:     $E \leftarrow$ the nodes directly supported by $c$ in CSTR
5:     for $e \in E$ do
6:         update $e$'s corresponding skylines
7:         $E \leftarrow$ the nodes directly supported by $e$
8:     end for
9: end for
```

VI. Evaluation

Our experimental studies attempt to answer two research questions: *i*) How is the performance of our approach compared with the state-of-the-art? *ii*) How do different problem scenarios influence the performance of our approach?

A. Experimental Setup

1) Dataset: We evaluate our approach using the QWS dataset[4], which is a public dataset crawled from the Web. The dataset contains 2,507 Web services, each described by nine QoS attributes measured using commercial benchmark tools.

2) Test Case: We use the service graph in Fig. 1 for our experiments, which includes seven functionalities. For each experiment, we randomly choose the concrete services from the QWS dataset to form the set of candidate services of each functionality and consider four QoS attributes, namely *response time*, *availability*, *throughput*, and *reliability*. We simply adopt the same utility function and constraints as [5] to define the optimization problem. For each request, we randomly designate one type of information from product specification, product review, and aggregated information, for it to retrieve. Therefore, there are totally eight execution paths for the three types of requests (shown by the examples in Section II).

3) Compared Methods: We compare the following methods:

- **BS:** basic skyline-based method that uses only functionality-level skylines.
- **DP:** our method described by Algorithm 1, using dynamic programming to determine which skylines to cache. The objective function in Eq. (1) is specifically defined as $\sum_{i=1}^{|CSTR^*|} h_i x_i$.
- **HA:** our method described by Algorithm 1, using heuristic method (described in Section IV-B) to cache skylines and adjusting skylines during usage.

We implemented all algorithms in Python and used the Gurobi solver[5] to solve the mixed integer programming model. All experiments were conducted on an Intel Core i7-5600 CPU

[4]http://www.uoguelph.ca/~qmahmoud/qws/

[5]http://www.gurobi.com/

with 32GB RAM, running on windows 10. Each experiment is run 100 times with the average values reported.

B. Experiments

Experiments 1-3 compare algorithms' online performance, while experiment 4 investigate both the online and offline performance.

1) Experiment 1: We compare the methods' performance on the same problem-scales under uniform request distribution, where the three types of requests are raised at equal frequency. We vary the number of candidate services of each functionality from 1,000 to 5,000, with the step-size of 1,000. Since BS always precompute seven skylines, although there are at most 12 skylines for the experimental process (Fig. 3a), we define $size_{max}$=7 to make a fair comparison. The algorithms are evaluated for every 500 requests. The experimental results are shown in the third column of Table II, where optimlaity is normalized to the interval of (0,1).

Results. DP and HA consistently achieve better optimality and require less computation time than BS. This conforms to our intuition that using skylines of larger granularities not only reduces the number of tasks but also decreases the number of candidate services. Reduction of search space leads to higher efficiency; since the reduction does not sacrifice accuracy, within a smaller search space, the solver has a higher chance of finding a better solution. Since DP and HA use the same heuristic, DP only slightly outperforms HA.

2) Experiment 2: We set the proportion of the three types of requests as follows, with all the other configurations same as Experiment 1: *i*) requests for product specification: 60%, *ii*) requests for aggregated info.: 30%, and *iii*) requests for product review: 10%. The experimental results are shown in the fourth column of Table II.

Results. DP and HA achieve significantly better performance in both optimality and efficiency than BS. It is easy to explain: while BS always follows the same routine regardless of the problem scenarios, DP and HA cache different skylines under different request distributions to utilize the limited cache size effectively. That means our methods can customize their caching solutions to fit the specific problem scenarios. The performance difference of DP and HA is not evident.

3) Experiment 3: Instead of varying the number of candidate services, we fix this number at 1,000 and vary the percentage of the services that have changes in their QoS. In particular, we randomly select a certain percentage of services and change their QoS values on a random attribute within the scope of $\pm$30% of its original values after every 50 requests. With all the other configurations same as Experiment 1, we obtain the experimental results shown in Fig. 4a.

Results. The efficiency of both DP and HA tends to decrease as more changes happen due to the higher cost of updating the subprocess-level skylines. The benefits keep outweighing the costs until the percentage of changed services becomes unrealistically large, i.e., 60%. The promising results can be attributed to the massively reduced repetitive computation regarding the recurring subprocesses. We omit the experiments on optimality, which is unaffected by the change in Qos.

4) Experiment 4: We study the influence of cache size on DP and HA by varying the cache size from 2 to 12, with the step-size of 2. We fix the number of candidate services at 1,000 and the percentage of changed services at 10%. Specially, we make sure the services' QoS changes after every 10 requests, so as to obtain an optimal cache size smaller than 12. The experimental results are shown in Fig. 4b.

Results. The efficiency of both DP and HA increases as the cache size grows, peaks at 8, and then decreases. That means over some point, the updating costs will outweigh the benefits. The optimality, in contrast, tends to monotonously increase. This is because, more cached skylines lead to a smaller search space, and in turn, a higher chance of the solver to find a better solution. The results indicate empirical adjustments to the cached skylines (like what we proposed in Section IV-B) is important, as the predefined cache size may not always be optimal.

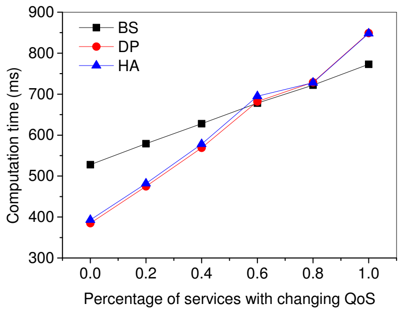
VII. Related Work

The skyline methods are used in service composition as a means of excluding redundant services from service selection [5]. For example, Yu *et al.* [6] introduce probabilistic dominant relationships to evaluate providers' performance and elicit users' preference. Since skylines are dependent on the QoS attributes concerned, Yang *et al.* [7] propose to accelerate skyline computation on a group of QoS attributes by leveraging the skylines on other combinations of QoS attributes. Berge *et al.* [8] propose a Block Nested Loop (BNL) algorithm for determining the dominance relation between candidate services to accelerate skyline computation. More recently, Wu *et al* [9] initially filter candidate services and then use so-called "abstraction refinement" technology to complete services selection, and Benouaret *et al* [10], further, propose the concept of regret constrained skyline, which is a relaxation of the constrained skyline, to avoid returning an empty result or miss interesting services. To reduce the number of skyline services, Alrifai *et al.* [5] propose to use representative skylines instead of skylines for service composition. Another direction in this field is to accelerate skyline computing by applying parallel computing infrastructure [11]. Until now, few works have focused on the multiple execution paths issue. Existing service composition methods commonly focus on the conditional branches in a service workflow, where the path to be executed can only be determined at runtime by some predefined conditions (e.g., in [12]). In contrast, the branches in service graphs are unconditional, meaning the optimal execution path need to be selected at design time.

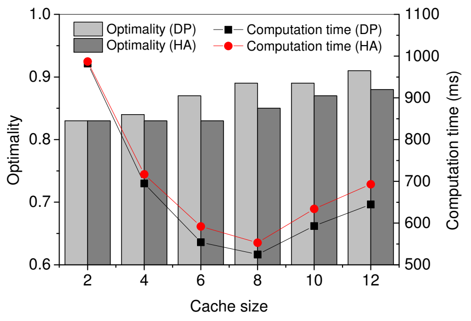
Our approach is based on two observations: first, applications of the same domain often have lots of similar workflows [13], [14]; second, different execution paths of the same service graph often share subprocesses (as shown by our motivating example). While traditional skyline-based methods always compute and reuse skylines at finest granularity—even when some subprocesses are repeatedly executed—their effect

TABLE II: Comparison under varying problem scales and request distributions

Metric	Method	Uniform distribution					Unbalanced distribution				
		1,000	2,000	3,000	4,000	5,000	1,000	2,000	3,000	4,000	5,000
Optimality	BS	0.81	0.82	0.78	0.82	0.83	0.87	0.84	0.80	0.78	0.79
	DP	0.89	0.87	0.88	0.85	0.90	0.89	0.88	0.81	0.78	0.81
	HA	0.85	0.84	0.82	0.85	0.89	0.89	0.87	0.80	0.78	0.81
Time(ms)	BS	528	924	1,238	1,338	1,654	429	795	1,095	1,142	1,312
	DP	385	753	943	1,189	1,286	198	431	652	689	719
	HA	393	752	951	1,210	1,292	202	428	653	692	723



(a) Efficiency w.r.t. QoS changes



(b) Optimality w.r.t. cache size

Fig. 4: Comparison of algorithms under varying rates of QoS change

are limited. According to Yu *et al.* [15], skylines over sets of services are smaller in size and can be computed efficiently and provided optimally as a package. In a similar way, we precompute, cache, and maintain the promising skylines following their structural relations in service graphs for their effective and efficient reuse in future compositions.

To our best knowledge, we are the first to consider varying levels of skylines for service composition. Du *et al.* [16] also compute subprocess-level skylines, but they focus on handling QoS correlations among candidate services. Other works similar to our study include: Hassan *et al.* [17] propose to cache the retrieved data to reduce data transmission of new mashups. Differing from that, we cache and reuse skylines instead of data. Tang *et al.* [18] discover patterns (i.e. set of services that are frequently used together along with their control flows) from previous composition records. Our approach also discovers patterns from historical compositions, but our patterns are merely abstract functionalities and subprocesses that serve as indexes for retrieving, updating, and reusing skylines. Although some efficient algorithms (e.g., [8]) are proposed for fast identification of skyline services for a fundamental functionality. Our work mainly concerns the update of skylines while the initial computation is less relevant. In addition, we develop formal procedures and algorithms to enable the reuse.

VIII. Conclusions

In this paper, we have proposed an approach of precomputing, caching, and reusing hierarchical skylines to reduce the computation load of determining the optimal service composition solution on service graphs. The problem is challenging as given multiple execution paths of a service graph, the two issues, i.e., the selection of the optimal execution path and the selection of the optimal constituent services, are correlated and should be addressed at the same time. We present the architecture, models, and algorithms to prepare, maintain, and reuse the cached skylines efficiently and economically. In particular, we selectively cache skylines using dynamic programming and heuristic methods and adjust them dynamically during usage to achieve the optimal effect. The lazy updating strategy ensures the updates are adaptive to the frequency and distribution of requests. Extensive experiments demonstrate the effectiveness of our approach. The encouraging results stimulate us to continue this study in the following aspects: *i*) incorporating request prediction techniques to increase reuse probability, *ii*) extending our approach to cope with more process structures like *loop*, and *iii*) exploring evolutionary caching strategies to enable real-time adaptation of our approach.

References

[1] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Info. Sci.*, vol. 280, pp. 218–238, 2014.

[2] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: A survey of techniques and tools," *ACM Comput. Surv.*, vol. 48, no. 3, p. 33, 2015.

[3] M. Razian, M. Fathian, R. Bahsoon, A. N. Toosi, and R. Buyya, "Service composition in dynamic environments: A systematic review and future directions," *Journal of Systems and Software*, p. 111290, 2022.

[4] F. Wagner, F. Ishikawa, and S. Honiden, "Qos-aware automatic service composition by applying functional clustering," in *Proc. ICWS*, 2011, pp. 89–96.

[5] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for qos-based web service composition," in *Proc. WWW*, 2010, pp. 11–20.

[6] Q. Yu and A. Bouguettaya, "Computing service skyline from uncertain qows," *IEEE Trans. Serv. Comput.*, vol. 3, no. 1, pp. 16–29, 2010.

[7] Y. Yang, F. Dong, and J. Luo, "Computing service skyeube for web service selection," in *Proc. CSCWD*, 2015, pp. 614–619.

[8] Y. Barge, L. Purohit, and S. Saha, “A skyline based technique for web service selection,” in *Smart Computing Techniques and Applications*. Springer, 2021, pp. 461–471.

[9] Z. Wu, K. Meng, X. Yan, D. Shi, and B. Hu, “Abstraction refinement approach for web service selection using skyline computations,” in *2021 IEEE World Congress on Services (SERVICES)*. IEEE, 2021, pp. 66–71.

[10] K. Benouaret, S. Elmi, and K.-L. Tan, “Relaxing the sky: Handling hard user constraints in skyline service selection,” in *2021 IEEE International Conference on Services Computing (SCC)*. IEEE, 2021, pp. 62–70.

[11] H. Liang, B. Ding, Y. Du, and F. Li, “Parallel optimization of qos-aware big service processes with discovery of skyline services,” *Future Generation Computer Systems*, vol. 125, pp. 496–514, 2021.

[12] 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, p. 6, 2007.

[13] J. Zhang, W. Tan, J. Alexander, I. Foster, and R. Madduri, “Recommend-as-you-go: a novel approach supporting services-oriented scientific workflow reuse,” in *Proc. SCC*, 2011, pp. 48–55.

[14] A. Goderis, P. Li, and C. Goble, “Workflow discovery: the problem, a case study from e-science and a graph-based solution,” in *Proc. ICWS*, 2006, pp. 312–319.

[15] Q. Yu and A. Bouguettaya, “Computing service skylines over sets of services,” in *Proc. ICWS*, 2010, pp. 481–488.

[16] Y. Du, H. Hu, W. Song, J. Ding, and J. Lu, “Efficient computing composite service skyline with qos correlations,” in *Proc. SCC*, 2015, pp. 41–48.

[17] O. A. H. Hassan, L. Ramaswamy, J. Miller *et al.*, “Mace: a dynamic caching framework for mashups,” in *Proc. ICWS*, 2009, pp. 75–82.

[18] R. Tang and Y. Zou, “An approach for mining web service composition patterns from execution logs,” in *Proc. WSE*, 2010, pp. 53–62.