Dynamic On-the-Fly Minimum Cost Benchmarking for Storing Generated Scientific Datasets in the Cloud

Dong Yuan, Xiao Liu, Member, IEEE, and Yun Yang, Senior Member, IEEE

Abstract—Massive computation power and storage capacity of cloud computing systems enable users to either store large generated scientific datasets in the cloud or delete and then regenerate them whenever reused. Due to the pay-as-you-go model, the more datasets we store, the more storage cost we need to pay, alternatively, we can delete some generated datasets to save the storage cost but more computation cost is incurred for regeneration whenever the datasets are reused. Hence, there should exist a trade-off between computation and storage in the cloud, where different storage strategies lead to different total costs. The minimum cost, which reflects the best trade-off, is an important benchmark for evaluating the cost-effectiveness of different storage strategies. However, the current benchmarking approach is neither efficient nor practical to be applied on the fly at runtime. In this paper, we propose a novel Partitioned Solution Space based approach with efficient algorithms for dynamic yet practical on-the-fly minimum cost benchmarking of storing generated datasets in the cloud. In this approach, we pre-calculate all the possible minimum cost storage strategies and save them in different partitioned solution spaces. The minimum cost storage strategy represents the minimum cost benchmark, and whenever the datasets storage cost changes at runtime in the cloud (e.g., new datasets are generated and/or existing datasets’ usage frequencies are changed), our algorithms can efficiently retrieve the current minimum cost storage strategy from the partitioned solution space and update the benchmark. By dynamically keeping the benchmark updated, our approach can be practically utilised on the fly at runtime in the cloud, based on which the minimum cost benchmark can be either proactively reported or instantly responded upon request. Case studies and experimental results based on Amazon cloud show the efficiency, scalability and practicality of our approach.

Index Terms—Cloud computing, minimum cost benchmarking, datasets storage and regeneration, scientific applications

1 INTRODUCTION

In recent years, cloud computing is emerging as the next generation computing paradigm which provides redundant, inexpensive and scalable resources on demand to users [10], [15]. Infrastructure as a Service (IaaS) is a very popular way to deliver services in the cloud [1], where users can deploy their applications in unified cloud resources such as computing and storage services without any infrastructure investments [35]. However, along with the convenience brought by using on-demand cloud services, users have to pay for the resources used according to the pay-as-you-go model, which can be substantial. Especially, nowadays applications are getting more and more data intensive [27], e.g., scientific applications [12], [13], [22], where the generated data are often gigabytes, terabytes, or even petabytes in size. These generated data contain important intermediate or final results of computation, which may need to be stored for reuse [8] and sharing [9]. Storing all the generated application datasets in the cloud may result in a high storage cost since some datasets (e.g., intermediate results) may be never reused but large in size. However, if we only store the most frequently used generated datasets in order to save the storage cost, whenever the deleted datasets are reused, we have to pay the computation cost for their regeneration. Hence, there is a trade-off between computation and storage for storing generated application datasets in the cloud. Based on this trade-off, different storage strategies are designed for the generated datasets in order to reduce the total application cost in the cloud [31], [33], [34].

The benchmark referred in this paper is the minimum cost for storage and regeneration of the datasets, which is used to evaluate the cost-effectiveness of storage strategies used in cloud applications. Due to the dynamic provisioning mechanism in cloud computing, this minimum cost varies from time to time whenever new datasets are generated or the datasets’ usage frequencies are changed. IaaS/Platform as a Service (PaaS) providers should be able to provide benchmarking services to Software as a Service (SaaS) providers in the cloud, who wish to know the minimum cost benchmark every now and then, so that they can evaluate the cost-effectiveness of the current storage strategies used for storing the generated datasets [32] for purposes of such as profit margin and market competitiveness. In this paper, by thoroughly investigating the issue of computation and storage trade-off, we describe a novel dynamic on-the-fly minimum cost benchmarking approach.
with highly efficient algorithms that can calculate the minimum cost for datasets storage in the cloud. In the approach we create a partitioned solution space (PSS), which saves all the possible minimum cost storage strategies (MCSS) of the datasets in the cloud. Therefore, whenever the application cost changes at runtime, our benchmarking algorithm can dynamically derive the new minimum cost from the PSS to keep the benchmark updated. Hence this approach can be utilised on the fly to either proactively report the dynamic minimum cost benchmark to SaaS providers or instantly respond to their benchmarking requests. Experimental results show the excellent efficiency, scalability and practicality of our approach.

The remainder of this paper is organised as follows. Section 2 analyses the research problems and presents some preliminaries. Section 3 presents our novel on-the-fly minimum cost benchmarking in detail. Section 4 evaluates our approach based on case studies and experimental results. Section 5 discusses the related work. Section 6 summarises our conclusions and points out future work. Due to the page limit, we have to put some important contents in supplementary materials, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2015.2389801, that are helpful for understanding this paper. For the ease of reading this paper, denotations are provided in supplementary materials Part 1, available online.

2 Problem Analysis and Preliminaries

As reported by Szalay and Gray in [27], science is in an exponential world and the amount of application data will double every year over the next decade and future. In supplementary materials Part 2, available online, we introduce two scientific applications, i.e. Astrophysics and Structural Mechanics respectively, as motivating examples for this paper. Both applications generate large datasets during execution, and users have to delete some valuable datasets due to storage limitation. In the cloud, the bottleneck of storage would not be the case because the commercial cloud service providers can offer virtually unlimited storage resources. However, due to the pay-as-you-go model, users are responsible for the cost of both storing and regenerating datasets in the cloud.

Based on the above background, we analyse the research problem in Section 2.1 and present some preliminaries from our prior work [32] in Section 2.2.

2.1 Problem Analysis

Theoretically, all of generated datasets can be stored or regenerated in the cloud. The trade-off between computation and storage is that the more datasets we store in the cloud, the more storage cost we need to pay, but when we need to reuse the datasets, the less computation cost we need to pay for their regeneration. On the contrary, the less datasets we store in the cloud, the less storage cost but more computation cost we need to pay for their regeneration. Due to the pay-as-you-go model, SaaS providers need to evaluate the cost effectiveness of their strategies for storing the increasing number of datasets generated in the cloud. Hence the IaaS/PaaS providers should be able and need to provide benchmarking services that can inform the minimum cost of storing the datasets, which is the best trade-off between computation and storage in the cloud. Finding the minimum cost storage strategy is a difficult problem because there could be a large number of datasets with complex dependencies between each other in the cloud. Our prior research [32] has made a theoretical breakthrough of solving this complicated problem with a recursive algorithm which is proved to be polynomial with the worst case time complexity of \(O(n^3)\) where \(n\) is the number of datasets. Due to the time complexity, this approach is unfortunately inefficient, which can only primarily be utilised for static (on-demand) minimum cost benchmarking, but impractical to be facilitated on the fly at runtime for applications.

In fact, the minimum cost of storing datasets in the cloud is a dynamic value. This is because of the dynamic nature of the cloud, i.e. 1) new datasets may be generated in the cloud at any time; and 2) the usage frequencies of datasets may also change as time goes on. Hence, the minimum cost benchmark may change from time to time. This requires the benchmarking algorithms being dynamic too in order to calculate the changing minimum cost on the fly at runtime. Furthermore, to guarantee the quality of service of response time in the cloud, the benchmarking algorithms also need to be efficient in order to instantly respond to requests of the minimum cost benchmark.

2.2 Preliminaries

2.2.1 Data Dependency Graph (DDG)

We use a data dependency graph [32] to represent the generated datasets in the cloud. DDG is a directed acyclic graph (DAG) that is based on data provenance in scientific applications. It can be built along with the application’s execution [17], [23]. Datasets defined in DDG are elementary (atomic) data used or generated by application tasks. One dataset can be used by different tasks from different users and generates different new datasets. In this paper, we assume that all the datasets once generated in the cloud, whether stored or deleted, their references are recorded in DDG. In other words, DDG depicts the generation relationships of datasets, with which the deleted datasets can be regenerated from their nearest stored preceding datasets. Fig. 1 shows a simple DDG, where every node in the graph denotes a dataset.

2.2.2 Datasets Storage and Regeneration Cost Model

In a commercial cloud computing environment, if the SaaS providers want to deploy and run applications, they need

1. Supplementary Materials, available online, are submitted with the paper.

2. The minimum cost storage strategy is not practical for runtime usage. Please refer to our prior work [33] for detailed discussions.

3. Please refer to Supplementary Materials Part 3, available online, for more details of DDG.
to pay for resources used. The resources are offered by IaaS/PaaS providers, which have their pricing models to charge the SaaS providers. In general, there are three basic types of resources in the cloud: storage, computation and bandwidth. Popular cloud services providers’ cost models are based on these types of resources. For example, Amazon cloud services’ prices are as follows:

- $0.15 per Gigabyte per month for the storage resources;
- $0.1 per CPU instance hour for the computation resources.

In this paper, we define our datasets storage and regeneration cost model in the cloud as follows:

\[
    \text{Cost} = \text{Computation} + \text{Storage},
\]

where \( \text{Cost} \) is the total cost for datasets storage and regeneration, \( \text{Computation} \) is the total cost of computation resources used to regenerate datasets, and \( \text{Storage} \) is the total cost of storage resources used to store datasets.

To utilise the datasets storage cost model, we define the attributes for the datasets in DDG the same as in [32]. Briefly, for a dataset \( d_i \in \text{DDG} \), its attributes are denoted as: \( <x_i, y_i, f_i, v_i, provSet_i, \text{CostR}_i> \), where

- \( x_i \) denotes the generation cost of dataset \( d_i \) from its direct predecessors in the cloud;
- \( y_i \) denotes the cost of storing dataset \( d_i \) in the cloud per time unit;
- \( f_i \) is a flag, which denotes the status whether dataset \( d_i \) is stored or deleted in the cloud;
- \( v_i \) denotes the overall usage frequency, which indicates how often \( d_i \) is used by different tasks.
- \( provSet_i \) denotes the set of stored provenance that are needed when regenerating dataset \( d_i \). Hence the generation cost of \( d_i \) is

\[
    \text{genCost}(d_i) = x_i + \sum_{\{k|d_k \in provSet_i \wedge d_k \rightarrow d_i\}} x_k. \tag{1}\n\]

- \( \text{CostR}_i \) is \( d_i \)'s cost rate, which means the average cost per time unit of dataset \( d_i \) in the cloud. The value of \( \text{CostR}_i \) depends on the storage status of \( d_i \), where

\[
    \text{CostR}_i = \begin{cases} y_i, & f_i = \text{stored} \\ \text{genCost}(d_i) \times v_i, & f_i = \text{deleted} \end{cases}. \tag{2}\n\]

Hence, the total cost rate of storing a DDG is the sum of \( \text{CostR} \) of all the datasets in it, which is \( \sum_{d_i \in \text{DDG}} \text{CostR}_i \).

We further define the storage strategy of a DDG as \( S \), where \( S \subseteq \text{DDG} \), which means storing the datasets in \( S \) in the cloud and deleting the rest. We denote the cost rate of storing a DDG with the storage strategy \( S \) as \( \text{SCR} \), where

\[
    \text{SCR} = \left( \sum_{d_i \in \text{DDG}} \text{CostR}_i \right)_S. \tag{3}\n\]

Based on the definition above, different storage strategies lead to different cost rates for the application. Hence, we use cost rate, i.e. cost per time unit, to evaluate the cost-effectiveness of the storage strategies for applications in the cloud. Our benchmarking algorithms aim at finding the storage strategy with the minimum cost rate.

2.2.3 CTT-SP Algorithm

The CTT-SP algorithm is proposed in our prior work [32]. It is a recursive algorithm that can find the minimum cost storage strategy for a general DDG. We have proven that it has a worst case time complexity of \( O(n^4) \), i.e. a polynomial complexity solution. The CTT-SP algorithm for linear DDG, i.e. a DDG with no branches where all the datasets in the DDG only have one predecessor and one successor except the start and end datasets, has a worst case time complexity of \( O(n^3) \), which we use in this paper to construct the new dynamic on-the-fly benchmarking approach. Here, we briefly introduce the CTT-SP algorithm for linear DDG in this section.

The essence of the CTT-SP algorithm is to construct a cost transitive tournament (CTT) based on the DDG. In the CTT, the paths from the start dataset to the end dataset have a one-to-one mapping to the storage strategies of the DDG, and the length of each path equals to the cost rate of the corresponding storage strategy. To satisfy this condition, for an edge from \( d_i \) to \( d_j \) in the CTT, we define its weight as the sum of cost rates of \( d_i \) and the datasets between \( d_i \) and \( d_j \), supposing that only \( d_i \) and \( d_j \) are stored and the rest of the datasets between \( d_i \) and \( d_j \) are all deleted. Formally,

\[
    \omega < d_i, d_j > = y_j + \sum_{\{d_k|d_k \in \text{DDG} \wedge d_i \rightarrow d_k \rightarrow d_j\}} (\text{genCost}(d_k) \times v_k),
\]

where \( \omega < d_i, d_j > \) denotes the weight of the edge from \( d_i \) to \( d_j \) in the CTT.

In Fig. 2, we demonstrate a simple example of constructing CTT by the CTT-SP algorithm for a DDG that only has three datasets.

Then we can use the well-known Dijkstra algorithm to find the shortest path from the start dataset to the end

8. Please refer to Supplementary Materials Part 5, available online, for more details of the linear CTT-SP algorithm.
dataset, where the datasets traversed by the shortest path form the minimum cost storage strategy of the DDG.

3 PSS Based Dynamic on-the-Fly Minimum Cost Benchmarking Approach

In this section, we describe our dynamic on-the-fly minimum cost benchmarking approach in detail. There are two simple assumptions in our approach: 1) the prices of cloud resources are fixed, and 2) the DDG and information of datasets are known (i.e., generation and storage cost, usage frequency) based on available history.

Section 3.1 describes the foundation of our benchmarking approach, which is a new concept of partitioned solution space. PSS is a novel mathematical model which has many useful properties. It saves all the possible minimum cost storage strategies of the DDG segment, which are calculated by the CTT-SP algorithm.

Section 3.2 describes the detailed benchmarking process. First, the minimum cost benchmark of the whole DDG can be calculated by saving and merging the PSSs. Next, whenever the minimum cost changes, the new benchmark can be dynamically derived on the fly from the pre-calculated PSSs by only calling the CTT-SP algorithm on the small DDG segment for adjustment. Hence we can keep the minimum cost benchmark updated on the fly so that SaaS providers’ benchmarking requests can be instantly responded.

3.1 Foundation of the Benchmarking Approach

In this paper, we utilise the linear CTT-SP algorithm to construct an entirely new approach for dynamic on-the-fly minimum cost benchmarking. The basic idea is that we divide the whole DDG into smaller linear DDG segments (DDG_LS) and create a partitioned solution space for every segment. The PSS saves all the possible minimum cost storage strategies of the DDG segment, which are calculated by the linear CTT-SP algorithm. Hence PSS is the foundation of our dynamic benchmarking approach presented in this paper, which will be carefully investigated in this section.

In this section, we first introduce the solution space of MCSSs and explain the reason why it exists for a DDG_LS (Section 3.1.1). Then we present some properties of the solution space (Section 3.1.2), based on which we develop the method for calculating PSS for a DDG_LS (Section 3.1.3). At last, we introduce the general form of the high dimension PSS for a DDG segment (Section 3.1.4).

3.1.1 MCSSs of a DDG_LS in a Solution Space

Generally speaking, a whole DDG only has one MCSS for storing the datasets in it, but for a DDG segment (e.g., DDG_LS), due to different preceding and succeeding datasets’ storage statuses, there would be different corresponding MCSSs, one for each status. In this section, we introduce a solution space that can save all the MCSSs.

The CTT-SP algorithm can be utilised on not only independent DDGs but also DDG_LSs, where the difference is the selection of start and end datasets for constructing the CTT. In the CTT-SP algorithm, for an independent DDG, we add two virtual datasets \( d_\text{s} \) and \( d_\text{e} \) as start and end datasets to construct the CTT. However, for the CTT of a DDG_LS, the start dataset \( d_\text{s} \) is the nearest stored preceding dataset to the DDG_LS, and the end dataset \( d_\text{e} \) is the nearest stored succeeding dataset to the DDG_LS. Fig. 3 shows an example of CTT for a DDG_LS.

Given different start and end datasets, the MCSS of a DDG_LS may be different. Hence, we need to analyse how the preceding and succeeding datasets impact the MCSS of the DDG_LS.

**Theorem 1.** For a DDG_LS, only the generation cost of its deleted preceding datasets and the usage frequencies of its deleted succeeding datasets impact its MCSS.

In a nutshell, Theorem 1 holds because: 1) the deleted preceding datasets have impact on the generation cost of datasets in the DDG_LS; 2) the regeneration of the deleted succeeding datasets needs to use datasets in the DDG_LS. Only these two factors impact the MCSS of the DDG_LS.

Based on Theorem 1, for a DDG_LS \( \{d_1, d_2, \ldots, d_\text{nl}\} \), we introduce two definitions:

- \( X = \sum_{i=1}^{\text{nl}} x_i \) is the sum of preceding datasets generation costs of a DDG_LS, where \( d_i \) is a deleted preceding dataset.
- \( V = \sum_{j=1}^{\text{nl}} y_j \) is the sum of succeeding datasets usage frequencies of a DDG_LS, where \( d_j \) is a deleted succeeding dataset.

For different start and end datasets of a DDG_LS, the values of \( X \) and \( V \) are different, and the MCSSs of the DDG_LS may also be different. Hence the MCSSs of a DDG_LS form a solution space. In other words, given different \( X \) and \( V \), there exist different MCSSs for storing the DDG_LS. As shown in Fig. 4, we denote an MCSS as \( S_{i,j} \), where \( d_i \) and \( d_j \) are the first and last stored datasets in the strategy, which could be any datasets in the DDG_LS. Conversely, any two datasets \( d_i \) and \( d_j \) in the DDG_LS may be the first and last stored datasets of an MCSS. Hence, theoretically, the number of different MCSSs for a DDG_LS is in the magnitude of \( n_i^2 \), where \( n_i \) is the number of datasets in the DDG_LS. In the following sections, we will describe how to find all the MCSSs for a DDG_LS and calculate the PSS.

9. Proofs of all the theorems in this paper are provided in Supplementary Materials Parts 6, 7, 12, and 13, available online.


3.1.2 MCSSs’ Cost Rates for a DDG_LS

In this section, we investigate the values of MCSSs’ cost rates in the solution space. Different MCSSs have different cost rates (i.e., SCR defined in formula (3) in Section 2.2.2) for storing the DDG_LS. Because DDG_LS is a segment of the whole DDG, the total cost rate of storing it includes not only the cost rate of itself, but also the cost rate of generating the deleted preceding and succeeding datasets. Hence, given any X and V, and the corresponding MCSS S_{i,j}, we denote the total cost rate of storing the DDG_LS \{d_1; d_2; \ldots; d_{nl}\} as TCR_{i,j}, where

\[ TCR_{i,j} = X \sum_{k=1}^{i-1} v_k + SCR_{i,j} + V \sum_{k=i+1}^{j} x_k. \]  

(4)

In formula (4), \( SCR_{i,j} \) is the cost rate of storing the DDG_LS with storage strategy \( S_{i,j} \), assuming that the direct preceding and succeeding datasets of DDG_LS be stored (i.e., an independent DDG). Formally,

\[ SCR_{i,j} = \left( \sum_{d_k \in DDG_LS} CostR_k \right) S_{i,j}. \]  

(5)

An important difference of \( TCR_{i,j} \) and \( SCR_{i,j} \) is that \( TCR_{i,j} \) is a variable for a storage strategy \( S_{i,j} \) depending on the value of \( X \) and \( V \) (see formula (4)), whereas \( SCR_{i,j} \) is a constant for a storage strategy \( S_{i,j} \) (see formula (5)).

For a DDG_LS, one extreme situation of \( (X = 0, V = 0) \) means that the start and end datasets are the direct preceding and succeeding datasets of the DDG_LS. Hence we can deem the DDG_LS as an independent DDG and directly call the CTT-SP algorithm to find its MCSS. In this situation, MCSS \( S_{u,v} \) found is the minimum \( SCR_{u,v} \) for storing the DDG_LS among other MCSSs, where \( TCR_{u,v} = SCR_{u,v} \). We denote \( S_{u,v} \) as \( S_{min} \) and \( SCR_{u,v} \) as \( SCR_{min} \).

The other extreme situation is that the start and end datasets are very far from the current DDG_LS, i.e. \( X > y_1/y_1, V > y_{nl}/x_{nl} \). Obviously, in this situation the first dataset \( d_1 \) and the last dataset \( d_{nl} \) in the DDG_LS should be stored. Hence we can deem \( d_1 \) and \( d_{nl} \) as the start and end datasets and call the CTT-SP algorithm for the datasets between \( d_1 \) and \( d_{nl} \). Strategy \( S \) found together with \( d_1 \) and \( d_{nl} \) form the MCSS of the DDG_LS in this situation denoted as \( S_{1,nl} \), where we also have \( TCR_{1,nl} = SCR_{1,nl} \). We denote \( S_{1,nl} \) as \( S_{max} \) and \( SCR_{1,nl} \) as \( SCR_{max} \).

Theorem 2. Given a DDG_LS \{d_1, d_2, \ldots, d_{nl}\}, \( SCR_{min} \) is the cost rate of MCSS \( S_{u,v} \) with \( X = 0, V = 0 \), and \( SCR_{max} \) is the cost rate of MCSS \( S_{1,nl} \) with \( X > y_1/y_1, V > y_{nl}/x_{nl} \). Then we have \( SCR_{min} < SCR_{i,j} < SCR_{max} \) where \( SCR_{i,j} \) is the cost rate of MCSS \( S_{i,j} \) with any given \( X \) and \( V \).

Theorem 2 tells us the two extreme situations: the lower and upper bounds for MCSSs of the DDG_LS, i.e. 1) the DDG_LS can be deemed as an independent DDG and 2) the first and last datasets in the DDG_LS should be stored. Fig. 5 shows the different MCSSs for a DDG_LS whose SCR values are in the valid range indicated in Theorem 2. We can further find all these strategies and save them in a strategy set, denoted as \( S_{All} \). The algorithm of finding \( S_{All} \) has the time complexity of \( O(n_t^4) \) where \( n_t \) is the number of datasets in the DDG_LS.

As discussed above, given any \( X \) and \( V \), there exists one MCSS for storing the DDG_LS in the set of \( S_{All} \). Hence we create coordinates of \( X \) and \( V \) to represent the solution space of all possible MCSSs for a DDG_LS. Furthermore we can calculate the distribution of the MCSSs in the solution space which is the PSS as described next.

3.1.3 PSS for Linear DDG (or DDG Segment)

In this section, we investigate the distribution of MCSSs in the solution space of a DDG_LS. We start with analysing the relationship of two MCSSs in the solution space and then describe the method for calculating the PSS of a DDG_LS.

We assume that \( S_{i,j} \) and \( S_{i',j'} \) be two MCSSs in \( S_{All} \) of a DDG_LS \{d_1, d_2, \ldots, d_{nl}\} and \( SCR_{i,j} < SCR_{i',j'} \). The border of \( S_{i,j} \) and \( S_{i',j'} \) in the solution space is that given particular \( X \) and \( V \), the total cost rates (TCR) of storing the DDG_LS with \( S_{i,j} \) and \( S_{i',j'} \) are equal. Hence we have

\[ TCR_{i,j} = TCR_{i',j'} \]

\[ X + \sum_{k=1}^{i-1} v_k + SCR_{i,j} + V + \sum_{k=i+1}^{j} x_k = X + \sum_{k=1}^{i-1} v_k + SCR_{i',j'} + V + \sum_{k=i+1}^{j} x_k \]

\[ X + \left( \sum_{k=1}^{i-1} v_k - \sum_{k=1}^{i} v_k \right) + \left( \sum_{k=i+1}^{j} x_k - \sum_{k=i+1}^{j+1} x_k \right) = 0 \]

(6)

From this equation we can see that the border of \( S_{i,j} \) and \( S_{i',j'} \) in the solution space is a straight line. Given different relationships of \( d_i \) and \( d_i', \) \( d_j \) and \( d_j' \), there are four different situations

1) \( d_i \rightarrow d_i' \wedge d_j \rightarrow d_j' \) as shown in Fig. 6a, formula (6) can be further simplified to:

\[ \sum_{k=i}^{i-1} v_k \times X + \sum_{k=i+1}^{j} x_k \times V + \left( SCR_{i,j} - SCR_{i',j'} \right) = 0. \]

(7)

2) \( d_i \rightarrow d_i' \wedge d_j \rightarrow d_j' \) as shown in Fig. 6b, formula (6) can be further simplified to:

\[ \sum_{k=i}^{i-1} v_k \times X - \sum_{k=i+1}^{j} x_k \times V + \left( SCR_{i',j'} - SCR_{i,j} \right) = 0. \]

(8)

10. Please refer to Supplementary Materials Part 8, available online, for the algorithm details.
Given the MCSSs, as long as \( d_i \) and \( d_j \) are positive values, \( S \) will never be in the solution space. We refer the eligible MCSSs in \( S_{All} \) as \( S_{ini} \) that is the initial input for calculating the solution space. The algorithm of eliminating \( S_{All} \) has the time complexity of \( O(n_3^2) \) where \( n_3 \) is the number of MCSSs in \( S_{All} \).

From the above discussion, we can see that the solution space of a DDG_LS is partitioned into different areas by lines. In a solution space, the MCSS of a DDG_LS changes from \( S_{min} \) to \( S_{max} \) as long as \( X \) and \( V \) increase. Given the MCSS set \( S_{ini} \), we calculate the partition line of every two strategies from \( S_{min} \) to \( S_{max} \) and gradually partition the solution space. Finally, we derive the partitioned solution space, which includes all the possible MCSSs of the DDG_LS. In the PSS, every area represents an MCSS and the partition lines are the borders.

An important property of partition lines in a PSS is that: if three MCSSs in the PSS are adjacent with each other, then the three partitions lines between every two MCSSs intersect at one point. Based on this property, we have designed the algorithm of calculating PSS12, which has a time complexity of \( O(n_3^3) \) where \( n_3 \) is the number of MCSSs in the PSS with the same magnitude of the number of MCSSs in \( S_{All} \). Fig. 7 demonstrates an example of the PSS of a DDG_LS with five MCSSs.

With the PSS, given any \( X \) and \( V \), we can locate the corresponding MCSS with the time complexity of \( O(n_3) \), where classical algorithms can be found in analytic geometry [28], hence we do not give detailed introduction in this paper.

3.1.4 PSS for a General DDG (or DDG Segment)

In this section, we present the method for calculating PSS of general DDGs (or DDG segments), which is the basis of our dynamic minimum cost benchmarking approach.

The PSS of a general DDG (or DDG segment) is a high dimension space, because the DDG may have branches where there may be more than one \( X \) and/or \( V \) value that determine the MCSS of the DDG. Although a general DDG’s PSS is different from the DDG_LS’s PSS, they have similar properties and can be calculated with similar algorithms.

Fig. 8 demonstrates an example of a DDG segment that has two branches. As we can see that because of two branches, the MCSS of the DDG segment is determined by three variables, which are \( X_1, V_2, V_3 \). Hence the solution space of this DDG segment is a three dimension space where every MCSS occupies some space. Similar to the solution space of DDG_LS, we can find the border of two MCSSs, which is a partition plane in the three dimension solution space. For example, we assume that \( S_{h,i,j} \) and
be two adjacent MCSSs in the solution space, where $SCR_{b_1,i,j} < SCR_{b_{i}',j'}$. The first and last stored datasets of these two strategies are in the positions as shown in Fig. 8.

The equation of the partition plane is

$$
\left( \sum_{k=0}^{h} v_k \right) * X_1 + \left( \sum_{k=0}^{v} x_k \right) * V_2 + \left( \sum_{k=0}^{f} x_k \right) * V_3
= SCR_{b_{i}',j'} - SCR_{b_1,i,j}.
$$

Two important properties of partition planes in a three dimension PSS are as follows:

1) For three MCSSs, if any two of them are adjacent with each other, then the three partition planes intersect in one line;

2) For four MCSSs, if any three of them intersect in a different line, then the four intersection lines intersect at one point.

By using these two properties,\(^{13}\) we can calculate the PSS of a DDG segment with two branches.

In a general DDG segment, there may exist multiple branches, hence there are more variables (i.e. more $X$ and $V$ dimensions) that impact the MCSS of the DDG segment. This makes the general DDG segment’s PSS a high dimension space, where the number of the dimensions is the total number of different $X$ and $V$ variables. For an $n$ dimension PSS, we assume that there be $m$ branches with preceding datasets (i.e. different $X$ dimensions), hence $n-m$ branches with succeeding datasets (i.e. different $V$ dimensions).

Given two MCSSs: 1) $S_p$ with the first stored datasets $d_{p,1}$, $d_{p,2}, \ldots, d_{p,n}$ in the $m$ different $X$ dimension branches and the last stored datasets $d_{p,(m+1)}$, $d_{p,(m+2)}, \ldots, d_{p,n}$ in the $n-m$ different $V$ dimension branches; 2) $S_q$ with the first stored datasets $d_{q,1}, d_{q,2}, \ldots, d_{q,n}$ in the $m$ different $X$ dimension branches and the last stored datasets $d_{q,(m+1)}, d_{q,(m+2)}, \ldots, d_{q,n}$ in the $n-m$ different $V$ dimension branches; and $SCR_p < SCR_q$, the border of $S_p$ and $S_q$ in the $n$ dimension space is:

$$
\sum_{i=1}^{m} \left( Bx_i * \left( \sum_{j=1}^{n} x_j \right) * X_i \right) + \sum_{j=m+1}^{n} \left( Bv_j * \left( \sum_{j=1}^{n} x_j \right) * V_j \right) = SCR_p - SCR_q,
$$

where $Bx_i = \begin{cases} -1, & d_{p,i} \rightarrow d_{q,i} \\ 0, & d_{p,i} = d_{q,i} \text{ and } Bv_j = \begin{cases} -1, & d_{q,j} \rightarrow d_{p,j} \\ 0, & d_{q,j} = d_{p,j} \\ 1, & d_{p,j} \rightarrow d_{q,j} \end{cases} \end{cases}$.

From the equation above, we can see that the border of two MCSSs in an $n$ dimension PSS is an $n-1$ dimension linear equation, which is an $n-1$ dimension space itself. In order to calculate the PSS of a general DDG segment, we need to investigate the intersections of the MCSSs in the $n$ dimension space.

**Theorem 3.** In an $n$ dimension PSS, for any $i$ MCSSs where $i \in \{2, 3, \ldots, (n+1)\}$, if any $(i-1)$ of the $i$ MCSSs intersect in a different $(n-i+2)$ dimension space, then the $i$ MCSSs intersect in an $(n-i+1)$ dimension space.

Theorem 3 describes the property of MCSS intersections in an $n$ dimension PSS, which is the generic form of the properties of partition lines and planes in the two and three dimension PSSs. We have proved this theorem using the linear equation theory. Based on Theorem 3, given the initial MCSS set of a general DDG segment (i.e. $S_{ini}$), we can design an algorithm to calculate the PSS in the similar way as the algorithm for calculating the PSS for DDG_LS. In Section 3.2, we will introduce how to derive $S_{ini}$ of a general DDG segment without calling the CTT-SF algorithm on it. For a PSS with $n_g$ dimensions, the border of MCSSs are $n_g$-variable linear equations and we need to solve the $n_g$-variable linear equations system to calculate an intersection point in the solution space which has a time complexity of $O(n_g^3)$ [24]. Hence the time complexity of calculating a general DDG segment’s PSS is $n_g^3$ times of the complexity for calculating the DDG_LS’s PSS, which is $O(n_2^3)$ (see Section 3.2.3), hence the total is $O(n_3^3 n_2^3)$.[5] Similarly, locating the MCSS in the high dimension PSS with given $X$ and $V$ values is also $n_g^3$ times more complex than locating an MCSS in the two dimension PSS, which is $O(n_1)$ (see end of Section 3.1.3), hence the total is $O(n n_2^3)$.

### 3.2 Dynamic on-the-Fly Minimum Cost Benchmarking

The reason that we calculate the PSS for a DDG segment is for dynamic minimum cost benchmarking. The philosophy of our approach is that we merge the PSSs of the DDG_LSs to derive the PSS of the whole DDG and save all the calculated PSSs along this process. From the PSS of the whole DDG, we can derive the minimum cost benchmark. Taking advantage of the pre-calculated results (i.e. the saved PSSs), whenever the application cost changes, we only need to recalculate the local DDG_LS’s PSS and quickly derive the new minimum cost benchmark for the whole DDG.

Section 3.2.1 describes how to derive the minimum cost benchmark, and Section 3.2.2 describes how to keep the benchmark updated.

#### 3.2.1 Deriving the Minimum Cost Benchmark

To calculate the minimum cost benchmark with our approach, we need to merge the DDG segments’ PSSs in order to derive the PSS of the whole DDG, from which we can locate the MCSS. To merge the PSSs of two DDG segments, we need to introduce another theorem.

**Theorem 4.** Given DDG segment $\{d_1, d_2, \ldots, d_m\}$ with PSS$_1$, DDG segment $\{d_{m+1}, d_{m+2}, \ldots, d_n\}$ with PSS$_2$, and the merged DDG segment $\{d_1, d_2, \ldots, d_m, d_{m+1}, d_{m+2}, \ldots, d_n\}$ with
PSS. Then we have:
\[
\forall S \in \text{PSS} \Rightarrow \left\{ \begin{array}{l}
S = S_1 \cup S_2, \quad S_1 \in \text{PSS}_1 S_2 \in \text{PSS}_2 \\
\text{SCR} = \text{SCR}_1 + \left( \sum_{k=j+1}^{n} x_k \right) \cdot \left( \sum_{k=m+1}^{n} v_k \right) + \text{SCR}_2
\end{array} \right.
\]
where \(d_j\) is the last stored dataset in the first DDG segment and \(d_i\) is the first stored dataset in the second DDG segment.

Theorem 4 tells us that:

1) The MCSSs in a larger DDG segment’s PSS (i.e. \(S\)) are combined by the MCSSs in its sub-DDG segments’ PSSs (i.e. \(S_1, S_2\)). Hence we can calculate the PSS of the larger DDG segment by merging the PSSs of its sub-DDG segments and do not need to call the CTT-SP algorithm on the larger DDG segment, which is inefficient.

2) The cost rate of the MCSS in the larger DDG segment (i.e. \(\text{SCR}\)) is the sum of cost rates of its sub-DDG segments’ MCSSs (i.e. \(\text{SCR}_1, \text{SCR}_2\)) and a parameter which is \(\left( \sum_{k=j+1}^{n} x_k \right) \cdot \left( \sum_{k=m+1}^{n} v_k \right)\). This parameter indicates the cost rate compensation for the datasets in the branches where the two sub-DDG segments merge together.

Fig. 9 further illustrates an example of Theorem 4 to merge two linear DDG segments.

Fig. 10 shows the pseudo code of merging two PSSs. In this algorithm, we first find the MCSS candidates set for the merged PSS (i.e. \(S_{\text{All}}\)) by combining the MCSSs in the two sub-PSSs (lines 1-7). During this process we also calculate the SCR for every MCSS (line 5) and find the upper bound for \(\text{SCR}_{\text{max}}\) (lines 6-7). Next, we eliminate the invalid MCSSs from \(S_{\text{All}}\), which includes two sub-steps: 1) deleting the MCSSs with invalid SCR values (lines 8-10); 2) calling the elimination algorithm\(^{14}\) to derive \(S_{\text{ini}}\) (line 11). Then we call the PSS calculation algorithm\(^{15}\) to calculate the PSS of the merged DDG segment (line 12). From the pseudo code, we can derive that the time complexity of merging two PSSs is the same as the calculation of the PSS, which is \(O(n^3)\).

To calculate the PSS of a general DDG in the cloud, we can calculate all the PSSs of its sub-DDG_LSs and gradually merge them to derive the PSS of the whole DDG. In order to achieve dynamic benchmarking, we need to save not only PSSs of the DDG_LSs, but also the PSSs calculated during the merging process. In our approach, we use a hierarchy data structure to save all the PSSs of a DDG, where an example of saving the PSS of a DDG with three sub-DDG_LSs is shown in Fig. 11.

In the PSS hierarchy, the level indicates the number of DDG_LSs merged in the PSS of the DDG segments at that level. For example, in Fig. 11, the DDG_LSs’ PSSs are saved at Level 1 of the hierarchy. Level 2, saves the PSSs of the DDG segments, which are connected by two DDG_LSs, e.g. \(\text{PSS}_{12}\) is the PSS of DDG segment combined by \(\text{DDG}_{LS_1}\) and \(\text{DDG}_{LS_2}\). Level 3 saves the PSS of the whole DDG, where we can see that the number of the hierarchy levels equals the number of DDG_LSs in the whole DDG. Furthermore, there are links among the levels in the hierarchy. A link between two PSSs at Levels \(i\) and \(i + 1\) means that the corresponding DDG segment of the PSS at Level \(i + 1\) contains the DDG segment of the PSS at Level \(i\). For example, in Fig. 11, there is a link between \(\text{PSS}_{1}\) and \(\text{PSS}_{12}\), because the DDG segment combined by \(\text{DDG}_{LS_1}\) and \(\text{DDG}_{LS_2}\) contains \(\text{DDG}_{LS_1}\).

The minimum cost benchmark and the corresponding MCSS can be derived from the PSS of the whole DDG, which is saved in the highest level of the hierarchy (e.g. Level 3 in Fig. 11). With this benchmark, we can either proactively report to SaaS providers or instantly respond to their benchmarking requests. In the next section, we will introduce how to dynamically keep this benchmark updated.

### 3.2.2 Updating the Minimum Cost Benchmark

Cloud is a dynamic environment. As time goes on, new datasets are generated in the cloud and the existing datasets’ usage frequencies may also change. Hence the minimum

14. Please refer to Supplementary Materials Part 9, available online, for details.

15. Please refer to Supplementary Materials Part 10, available online, for details.
cost benchmark of storing the datasets would also change accordingly. By taking the advantage of the PSS hierarchy, we can dynamically calculate the new minimum cost benchmark on-the-fly whilst storing the datasets. There are two situations to be dealt with:

1) New datasets are generated in the cloud. The algorithm pseudo code of calculating the new minimum cost benchmark of this situation is shown in Fig. 12. Assuming that the new datasets be in a DDG_LS (if not, we take its sub-DDG_LS), first we add it to the whole DDG and calculate its PSS, denoted as PSS_new (lines 1-3). Next, for every MCSS in PSS_new, we locate the corresponding MCSS from the original DDG’s PSS (lines 4-7) and calculate the cost rate of the whole DDG, i.e. SCR (line 8). Then, we find the minimum SCR as the new minimum cost benchmark for the whole DDG and the corresponding storage strategy as the new MCSS (lines 9-11). In this whole process, we only need to calculate the PSS of the new DDG_LS, which is usually small in size, and the PSS of the original DDG has already been pre-calculated and saved in the hierarchy. Hence we can quickly update the minimum cost benchmark. For example, in Fig. 13a, for new DDG_LS4, we calculate PSS4 and connect it with existing PSS123 in the hierarchy to derive the new minimum cost benchmark and the MCSS of the whole DDG.

After calculating the new minimum cost benchmark, we have to update the PSS hierarchy for the new DDG_LS. For every newly added PSS at Level i of the hierarchy (starting from Level 1 to the highest level), we find its connected DDG_LS in the whole DDG and connect them to form a new segment. We calculate the PSS of the new segment and add it to Level i + 1 of the hierarchy as well as the corresponding links between the two levels. An example of updating the PSS hierarchy in this situation is shown in Fig. 13a, where the shadowed PSSs are the new ones added to the hierarchy after adding PSS4.

2) Existing datasets’ usage frequencies are changed. In this situation, we first find the DDG_LS that contains the datasets whose usage frequencies are changed. As shown in the pseudo code in Fig. 14, we also need to calculate the DDG_LS’s PSS at the beginning (lines 1-3). Then, we find the PSSs of the rest parts of the whole DDG except the changed DDG_LS and save them in a set, i.e. PSS_Set (line 4). Next, for every MCSS in the new PSS (line 6), we calculate the X and V values (line 7) to locate the corresponding MCSSs of the DDG segments that are connected to the changed DDG_LS from PSSs in PSS_Set (lines 8-17). We also calculate the corresponding cost rate of the whole DDG, i.e. SCR (line 18). Then, we find

Algorithm: Update benchmark (usage frequencies changed)
Input: DDG_LS //With the changed datasets
The PSS hierarchy
Output: S //MCSS of the whole DDG
SCR //New minimum cost benchmark

01. S_All = Find S_All (DDG_LS);
02. S_init = Eliminate S_All (S_init); //O(n^2)
03. PSS_new = Calculate PSS (S_init); //O(n^2n^2)
04. SCR = ∞; S = φ;
05. for (every S_i in PSS_new) do:
06. \[ V = \sum_{k=1}^{i} v_k; \]
07. \[ S_{tmp} = PSS_{Locate}(0...0,V); \]
08. \[ SCR_{new} = SCR_{tmp} + \left( \sum_{k=1}^{i} x_k \right) \cdot V + SCR_{i,j}; \]
09. if (SCR > SCR_{new}) do:
10. \{ S = S_{tmp} ; S_{i,j} ; \}
11. \{ SCR = SCR_{new} ; \}
12. Return S, SCR;

Fig. 14. Pseudo code for calculating new minimum cost benchmark when datasets’ usage frequencies are changed.
the minimum SCR as the new minimum cost benchmark for the whole DDG and the corresponding storage strategy as the new MCSS (lines 19-20). In this whole process, when calculating the PSS of the changed DDG_LS, we only update the weights of some edges in the existing CTT and do not need to create a new one. Furthermore, the PSSs of DDG segments in PSS_Set have already been pre-calculated and saved in the hierarchy. Hence we can quickly update the minimum cost benchmark. For example, in Fig. 13b, we re-calculate PSS_2 for changed DDG_LS_2. To derive the new minimum cost benchmark for the whole DDG, we connect new PSS_2 with PSS_13 and PSS_1 in PSS_Set, which are the rest parts of the whole DDG except DDG_LS_2.

After calculating the new minimum cost benchmark, we also need to update the PSS hierarchy for the changed PSS. For every changed PSS at Level i of the hierarchy (starting from Level 1 to the highest level), we find the PSSs at Level i + 1 that are linked with it and update all these PSSs. An example of updating the PSS hierarchy in this situation is shown in Fig. 13b, where the shadowed PSSs are the new ones that we need to update in the hierarchy after changing of PSS_2.

Based on the above algorithms in this section, in terms of efficiency, our approach can instantly respond to SaaS providers’ benchmarking requests by keeping the minimum cost benchmark updated on the fly. Whenever new datasets are generated and/or existing datasets’ usage frequencies are changed, our algorithm can quickly calculate the new minimum cost benchmark in $O(n^2 d_3)$ (see Figs. 12 and 14), where all the parameters are for the local DDG_LS, which are usually very small. The total time complexity of our benchmarking approach includes updating the hierarchy, which is used to save the PSSs. We use $m$ to denote the number of DDG_LS in the whole DDG. In the situation of new datasets generation, we need to add one new PSS to every level of the PSS hierarchy (see Fig. 13a), where the number of the levels equals to the number of DDG_LSs in the whole DDG, hence the time complexity is $O(mn^3 d_3)$. In the situation of existing datasets’ usage frequencies changing, we have to recalculate more than one PSS (i.e. in the magnitude of $m$) in every level of the hierarchy (see Fig. 13b), hence the time complexity is $O(m^2 n^3 d_3)$. In Section 4, we will use experimental results to further demonstrate the efficiency as well as the scalability of our approach.

4 Evaluation

The cost model of our novel PSS based dynamic on-the-fly minimum cost benchmarking approach is generic; hence it can be used in any applications with different price models for cloud services. In this section, we evaluate our approach from two aspects based on Amazon cloud.

In Section 4.1, we present case studies of the motivating examples addressed in supplementary materials Part 2, available online, which demonstrate how our approach can be utilised in specific real world scientific applications. Due to the page limit, we present the pulsar searching application in this section and the finite modelling application in supplementary materials Part 15, available online.

In Section 4.2, we present the random experimental results to evaluate the general performance of our benchmarking approach, such as efficiency, scalability, practicality and utilisation.

4.1 Case Study of Pulsar Searching Application

Swinburne Astrophysics group has been conducting pulsar surveys in Swinburne supercomputer using the observation data from Parkes Radio Telescope, which is one of the most famous radio telescopes in the world. The application contains complex and time-consuming tasks and needs to process gigabytes, even terabytes of data that are streamed from the telescope at the rate of 1 GB/s. During the execution, datasets of large size are generated. As time goes on, users may need to reanalyse some datasets, or reuse them for new analyses where more datasets are generated. At present, all these datasets are deleted after being used once due to the storage limitation of the supercomputer. In this section, we demonstrate how our benchmarking approach is utilised for storing generated datasets, if the application is deployed in the cloud.

To demonstrate the cost effectiveness of our minimum cost benchmark, we compare it with some representative storage strategies as follows: 1) store all datasets strategy; 2) store none dataset strategy; 3) usage based strategy, in which we store the datasets that are most frequently used; 4) generation cost based strategy, in which we store the datasets that incur the highest generation costs; 5) cost rate based strategy reported in [34], in which we store the datasets by comparing their own generation cost rates and storage cost rates; and 6) local-optimisation based strategy reported in [33], in which the CTT-SP algorithm is utilised to achieve a localised optimum for storing a large DDG.

In this case, we derive the minimum cost benchmark for storing the datasets generated in one branch of the pulsar searching workflow for processing one hour’s observation data. Then, by using the benchmark, we evaluate the cost-effectiveness of different strategies for storing datasets generated by the pulsar searching application for processing one day’s observation data.

The execution of the application has two main phases: Files Preparation and Seeking Candidates, where in each phase three datasets are generated. For illustration, a DDG segment generated in this application is shown in Fig. 15, as well as the sizes and generation times of the datasets. We assume that the prices of cloud services also follow Amazon clouds’ cost model. From Swinburne astrophysics research group, we understand that the “de-dispersion files” are the most useful datasets. Based on these files, many accelerating and seeking methods can be used to search pulsar candidates. Based on the scenario, we set the “de-dispersion files”
Fig. 15. PSS of a pulsar searching DDG segment.

to be used once every 4 days, and the rest of the datasets to be used once every 10 days.

Fig. 15 also demonstrates the benchmarking process of PSSs calculation for the two DDG_LSs and the merging process for the PSS of the whole DDG segment generated (Section 3.2.1). As shown in Fig. 15, when datasets \( d_1, d_2, d_3 \) are generated as \( DDG_LS_1 \), we calculate \( PSS_1 \) which contains four MCSSs. Next, when datasets \( d_4, d_5, d_6 \) are generated as \( DDG_LS_2 \) we calculate \( PSS_2 \) which contains only one MCSS. By merging \( PSS_1 \) and \( PSS_2 \), we can get the PSS of the whole DDG segment, which contains two MCSSs.

From the merged PSS, we can derive the minimum cost benchmark of the DDG segment, which is $0.51 per day for storing these six datasets and the corresponding MCSS is storing datasets \( d_2, d_4, d_6 \) only (\( d_1, d_3, d_5 \) are deleted). After we get the new benchmark, all the calculated PSSs are saved in the hierarchy for further use.

Based on our approach, we can also derive the minimum cost benchmark for storing the datasets generated in the whole application and for processing one day’s observation data, which is $159.12 per day. With this minimum cost benchmark, we can evaluate the cost-effectiveness of different storage strategies working on the pulsar searching application. Table 1 shows the comparison of cost rates of the representative storage strategies and the minimum cost benchmark for storing the generated datasets for processing one day’s observation data.

From Table 1 we can see that in this case 1) the local-optimisation based strategy achieves the minimum cost for storing the generated datasets; 2) the cost rate based strategy is also quite cost effective; 3) the rest four strategies are not cost effective comparing to the minimum cost benchmark.

---

19. Please refer to Supplementary Materials Part 14, available online, for more details about how datasets are stored with different strategies and their daily cost.

20. ECU (EC2 Computing Unit) is the basic unit defined by Amazon to measure the compute resources. Please refer to the following address for details. http://aws.amazon.com/ec2/instance-types/

21. Please refer to Supplementary Material Parts 16–17, available online, for more details.

22. Please refer to Supplementary Materials Part 16, available online, for the impact of the parameters.

23. Amazon cloud service offers different CPU instances with different prices, where using expensive CPU instances with higher performance would reduce computation time. There exists a trade-off of time and cost [16] which is different to the trade-off of computation and storage described in this paper, hence is out of this paper’s scope.
updated benchmark of the whole DDG with both the on-demand benchmarking approach and our new dynamic on-the-fly benchmarking approach. As both approaches would derive the same minimum cost benchmark for the same DDG, we focus on their efficiency and scalability for practicality. Fig. 16 shows the comparison of CPU time consumed by the two benchmarking approaches on Amazon EC2 standard small instance (m1.small).

From Fig. 16 we can see that the on-demand benchmarking approach is neither efficient nor scalable to keep the minimum cost benchmark updated at runtime. The computation time increases dramatically as the datasets number increases. This is because whenever the cost is changed in the cloud, either because of the new datasets generation or the changes of existing datasets’ usage frequencies, we need to call the CTT-SP algorithm for the whole DDG to calculate the new benchmark.

In contrast, for the dynamic benchmarking approach, as we can see from the zoom-in chart (bottom plane) in Fig. 16, the time for calculating new minimum cost benchmark is in the magnitude of seconds in general, hence very efficient; and it increases marginally even when the number of datasets in the DDG increases significantly, hence very scalable. This is because we take advantage of the pre-calculated PSSs that are saved in the hierarchy and only need to recalculate the PSS of the local DDG_LS to derive the new benchmark. Hence, the complexity of calculating the new benchmark is more or less independent of the size of the DDG. In fact, our algorithms can dynamically keep the minimum cost benchmark updated (at background), so that we can normally guarantee an instant response to SaaS providers’ benchmarking requests.

More specifically, the zoom-in chart (bottom plane) in Fig. 16 shows that the time for calculating new minimum cost benchmark in the situation of datasets’ usage frequencies changing is less than new datasets generation. This is because when new datasets are generated, we need to create a new CTT for them to calculate the new PSS, whereas when existing datasets’ usage frequencies change in a DDG_LS, we only need to update the weights of the changed edges in the existing CTT instead of creating a new one to recalculate the PSS.

From Fig. 16, we also note that after the calculation of new benchmark, the update of the PSS hierarchy takes some computation time, which can again be a background job. However, even taking the update of the PSS hierarchy into account, the total computation time of our dynamic on-the-fly benchmarking approach is still much less than the on-demand approach, not mention that the on-demand approach is a foreground job whilst the SaaS provider waits for the response. This is because the complexity of updating the PSS hierarchy in the new approach only depends on the number of DDG_LSs whereas the on-demand approach depends on the number of datasets in the whole DDG.

The experimental results of Fig. 16 are consistent with the theoretical results presented in the final paragraph of Section 3.2.2. More specifically from Fig. 16 we can see that the computation time of updating the PSS hierarchy for datasets’ usage frequencies changing increases slowly as the number of DDG_LS grows, while in the situation of new datasets generation, the computation time increases much slower which is in a linear manner. This is because the newly generated datasets only have preceding datasets in the original DDG, while the corresponding DDG_LS of the datasets whose usage frequencies are changed has both preceding and succeeding datasets in the original DDG. According to the rules of updating the PSS hierarchy introduced in Section 3.2.2 (see Fig. 13), we have to recalculate more PSSs in the hierarchy for the datasets’ usage frequencies changing situation than the new datasets generation situation.

TABLE 1

<table>
<thead>
<tr>
<th>Stored dataset</th>
<th>Deleted datasets</th>
<th>Cost Rate ($/day)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store all datasets</td>
<td>$d_1, d_2, d_3, d_4, d_5, d_6$</td>
<td>$d_1, d_2, d_3, d_4, d_5, d_6$</td>
</tr>
<tr>
<td>Store none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Usage based strategy</td>
<td>$d_2$</td>
<td>$d_1, d_3, d_4, d_5, d_6$</td>
</tr>
<tr>
<td>Generation cost based strategy</td>
<td>$d_2$</td>
<td>$d_1, d_3, d_4, d_5, d_6$</td>
</tr>
<tr>
<td>Cost rate based strategy</td>
<td>$d_2$ (deleted initially), $d_4, d_6$</td>
<td>$d_1, d_3, d_5$</td>
</tr>
<tr>
<td>Local-optimisation based strategy</td>
<td>$d_2, d_4, d_6$</td>
<td>$d_1, d_3, d_5$</td>
</tr>
<tr>
<td>Minimum cost benchmark</td>
<td>$d_2, d_4, d_6$</td>
<td>$d_1, d_3, d_5$</td>
</tr>
</tbody>
</table>

From Fig. 16 we can see that the on-demand benchmarking approach is neither efficient nor scalable to keep the minimum cost benchmark updated at runtime. The computation time increases dramatically as the datasets number increases. This is because whenever the cost is changed in the cloud, either because of the new datasets generation or the changes of existing datasets’ usage frequencies, we need to call the CTT-SP algorithm for the whole DDG to calculate the new benchmark.

In contrast, for the dynamic benchmarking approach, as we can see from the zoom-in chart (bottom plane) in Fig. 16, the time for calculating new minimum cost benchmark is in the magnitude of seconds in general, hence very efficient; and it increases marginally even when the number of datasets in the DDG increases significantly, hence very scalable. This is because we take advantage of the pre-calculated PSSs that are saved in the hierarchy and only need to recalculate the PSS of the local DDG_LS to derive the new benchmark. Hence, the complexity of calculating the new benchmark is more or less independent of the size of the DDG. In fact, our algorithms can dynamically keep the minimum cost benchmark updated (at background), so that we can normally guarantee an instant response to SaaS providers’ benchmarking requests.

More specifically, the zoom-in chart (bottom plane) in Fig. 16 shows that the time for calculating new minimum cost benchmark in the situation of datasets’ usage frequencies changing is less than new datasets generation. This is because when new datasets are generated, we need to create a new CTT for them to calculate the new PSS, whereas when existing datasets’ usage frequencies change in a DDG_LS, we only need to update the weights of the changed edges in the existing CTT instead of creating a new one to recalculate the PSS.

From Fig. 16, we also note that after the calculation of new benchmark, the update of the PSS hierarchy takes some computation time, which can again be a background job. However, even taking the update of the PSS hierarchy into account, the total computation time of our dynamic on-the-fly benchmarking approach is still much less than the on-demand approach, not mention that the on-demand approach is a foreground job whilst the SaaS provider waits for the response. This is because the complexity of updating the PSS hierarchy in the new approach only depends on the number of DDG_LSs whereas the on-demand approach depends on the number of datasets in the whole DDG.

The experimental results of Fig. 16 are consistent with the theoretical results presented in the final paragraph of Section 3.2.2. More specifically from Fig. 16 we can see that the computation time of updating the PSS hierarchy for datasets’ usage frequencies changing increases slowly as the number of DDG_LS grows, while in the situation of new datasets generation, the computation time increases much slower which is in a linear manner. This is because the newly generated datasets only have preceding datasets in the original DDG, while the corresponding DDG_LS of the datasets whose usage frequencies are changed has both preceding and succeeding datasets in the original DDG. According to the rules of updating the PSS hierarchy introduced in Section 3.2.2 (see Fig. 13), we have to recalculate more PSSs in the hierarchy for the datasets’ usage frequencies changing situation than the new datasets generation situation.
The experimental results in Fig. 16 show the high efficiency and scalability of our novel PSS based benchmarking approach. This is because our benchmarking approach is based on the pre-calculated PSSs, and the efficiency of calculating PSS depends on the number of MCSSs in the PSS rather than the number of datasets in the DDG.  

4.2.4 Practicability Evaluation

As described in Fig. 16, after quickly deriving the new minimum cost benchmark whenever the application cost changes in the cloud, the background job of updating the PSS hierarchy may take some time. In some extreme cases, e.g. the number of datasets is very large or there are very frequent benchmarking requests from SaaS providers, we can further reduce this background computation time via parallelisation. To demonstrate the practicability of our approach, we have developed a distributed version of the proposed algorithms and deployed them on multiple Amazon EC2 instances.

To evaluate the effect of parallelisation for our approach, we use different number of EC2 instances to calculate the PSS hierarchy of a randomly generated DDG with 200 datasets and four DDG LSs (without losing generality). Because the PSS calculation is mainly computation-oriented, we choose to use the High CPU EC2 instances with Amazon Linux Image as follows: 1) cl.xlarge instance with eight cores (20 ECUs) and 2) cl.medium instance with two cores (five ECUs). The experimental results are shown in Fig. 17.

As shown in Fig 17, when we use more EC2 instances to calculate the PSS hierarchy, the computation time decreases proportionally. It indicates that the parallelisation can greatly reduce the response time of our approach. Hence, we deem that our approach is practical in cloud environments.

4.2.5 Utilisation of Benchmarking

The minimum cost benchmark can be used to evaluate the cost-effectiveness of strategies for storing randomly generated DDGs. The cost rate comparison of representative storage strategies with the minimum cost benchmark is shown in Fig 18.

We can clearly see that 1) store none and store all datasets are the least cost-effective strategies; 2) the cost rate based strategy and the local-optimisation based strategy are more cost effective than the usage based strategy and the generation cost based strategy. Especially, the local-optimisation based strategy is the most cost-effective one whose cost rate is very close to the benchmark. From the comparison, we can see that our benchmark can be utilised easily to evaluate the cost-effectiveness of specific storage strategies.

4.3 Summary

Base on the experiments presented in this entire section, we can come to the conclusion that our novel PSS based dynamic minimum cost benchmarking approach is highly efficient, scalable and practical to be facilitated on the fly at runtime in the cloud.

5 Related Work

Today, research on deploying applications in the cloud becomes popular [20]. Cloud computing system for scientific applications, i.e. science cloud, has already commenced [2], [3], [4]. This paper is mainly inspired by the work in two research areas: cache management and scheduling. With smart caching mechanism [19], system performance can be greatly improved. The similarity is that both pre-store some data for future use, while the difference is that caching is to reducing data accessing delay but our work is to find the minimum application cost. Some works in scheduling focus on reducing various costs for either applications [26], [36] or systems [30]. However these works mainly focus on resource utilisation rather than the perspective of the trade-off between computation and storage. This trade-off is a unique issue in the cloud due to the pay-as-you-go model, hence our benchmarking approach is brand new which is different from other existing benchmarking counterparts in the cloud [20], [11].

Researchers are exploring the cost-effectiveness of the cloud, because comparing to the traditional computing systems like cluster and grid, a cloud computing system has a cost benefit in various aspects [6]. Assunção et al. [7] demonstrate that cloud computing can extend the capacity of clusters with a cost benefit. With Amazon clouds’ cost model and
BOINC volunteer computing middleware, the work in [21] analyses the cost benefit of cloud computing versus grid computing. The work by Deelman et al. [13] also applies Amazon clouds’ cost model and demonstrates that cloud computing offers a cost-effective way to deploy scientific applications. The above works mainly focus on the comparison of cloud computing systems and the traditional computing paradigms, which shows that applications running in the cloud have cost benefits, but they do not touch the issue of computation and storage trade-off for datasets in the cloud.

An important foundation for investigating the issue of computation and storage trade-off is the research on data provenance. Due to the importance of data provenance in scientific applications, many works about recording data provenance of the system have been done [17]. Recently, research on data provenance in cloud computing systems has also appeared [23]. More specifically, Osterweil et al. [25] present how to generate a data derivation graph for the execution of a scientific workflow, where one graph records the data provenance of one execution, and Foster et al. [14] propose the concept of Virtual Data in the Chimera system, which enables the automatic regeneration of datasets when needed. Our DDG is based on data provenance in scientific applications, which depicts the dependency relationships of all the datasets in the cloud. With DDG, we know where the datasets are derived from and how to regenerate them.

Nectar system [18] is designed for automatic management of data and computation in data centers, where obsolete used datasets are deleted and regenerated whenever reused in order to improve resource utilisation. In [13], Deelman et al. present that storing some popular intermediate data can save the cost in comparison to always regenerating them from the input data. In [5], Adams et al. propose a model to represent the trade-off between computation cost and storage cost, but have not given the strategy to find this trade-off. The work reported in [31], [33], [34], proposes practical cost-effective strategies for datasets storage of scientific applications, but it is not the minimum cost storage strategy in the cloud. In [32], Yuan et al. propose the CTT-SP algorithm with a polynomial time complexity (i.e. $O(n^3)$) that can calculate the minimum cost of storing scientific datasets in the cloud with fixed usage frequencies. However, this algorithm is only suitable for static scenarios. Whenever datasets’ usage frequencies are changed or new datasets are generated, we have to run the CTT-SP algorithm on all datasets stored in the cloud to calculate the new benchmark, which is not efficient. Hence, it can only be utilised for on-demand minimum cost benchmarking.

In this paper, we propose an entirely new approach for dynamic on-the-fly minimum cost benchmarking of datasets storage in the cloud. Significantly different from the on-demand benchmarking approach [32], we divide the DDG into small segments and pre-calculate all the possible minimum cost storage strategies (i.e. the solution space) of every DDG segment using the CTT-SP algorithm. By utilising the pre-calculated results, whenever new datasets are generated and/or existing datasets’ usage frequencies are changed, we develop efficient algorithms that can dynamically calculate the changing minimum cost benchmark at runtime by calling the CTT-SP algorithm only on the local DDG segment. By keeping the minimum cost benchmark updated on-the-fly, SaaS providers’ benchmarking request can be instantly responded. Hence, it is a practical approach for dynamic minimum cost benchmarking of storing generated datasets in the cloud. The experiments in Section 4 have shown the efficiency and scalability of the PSS based dynamic benchmarking algorithms comparing to the original CTT-SP algorithm for on-demand benchmarking reported in [32].

6 CONCLUSIONS AND FUTURE WORK

In this paper, based on an astrophysics pulsar searching scenario, we have examined the unique issues of storing application datasets in the cloud and analysed the requirements of dynamic on-the-fly minimum cost benchmarking. We have proposed a novel partitioned solution space based practical approach with innovative algorithms that can dynamically calculate the minimum cost benchmark for storing generated application datasets in the cloud, which achieves the best trade-off between computation cost and storage cost of the cloud resources. Both theoretical analysis and experimental results demonstrate that our novel dynamic minimum cost benchmarking approach is highly efficient and scalable. Hence, it can be practically utilised on the fly at runtime in the cloud, which was unavailable before.

Our current work is based on Amazon clouds’ cost model and assumes that all the application data be stored with a single cloud service provider. However, sometimes large-scale applications have to run in a more distributed manner since some application data may be distributed with fixed locations and the application can also be deployed in resources with different performance. In the future, we will incorporate the data transfer cost and more complex pricing models into our minimum cost benchmarking.

ACKNOWLEDGMENTS

The research work reported here is partly supported by Australian Research Council under DP110101340 and LP130100324. The authors are grateful for the discussions on pulsar searching process with Dr. W. van Straten and Ms. L. Levin and on the Finite Element Modelling application with Dr. S. Xu, from Faculty of Science, Engineering and Technology, Swinburne University of Technology. They are also grateful to Mr. T. Wiens for the assistance in experiments and to Mr. A. Giardina for English proofreading. Y. Yang is the corresponding author of the paper.

REFERENCES


