TRADING STORAGE FOR COMPUTATION

By

Jis Ben
Clement Lee
Bhumi Thakkar

TABLE OF CONTENTS

1. Abstract

The orthodox programming model thus far has involved only two steps, computation and storage. Regardless of the frequency of usage of a dataset or the size of the dataset, all results are labeled for storage under such a scheme. This 'Store-All' scheme results in higher costs than necessary when large, rarely used datasets are stored. Recent technologies with large, flexibly allocable resources such as cloud based storage systems lead to a possible alternative to this traditional paradigm. The question thus becomes one of when choosing to re-compute a dataset will prove to be more cost-effective than simply storing the dataset. This project attempts to combine factors such as reuse rate, real world cost figures from Amazon Web Services, and data I/O costs to determine when this storage/re-computation tradeoff should be taken. A comparison is made with the traditional 'Store-All' method to illustrate that it is possible to reduce the costs associated with managing a large dataset in a cloud based storage system.

2. Introduction

The recent rise of the cloud and its massive computational power calls into question the Standard computing norm of compute, store and recall. The cloud provides resources that can be allocated on demand to re-compute data-sets very quickly instead of waiting on I/O to retrieve data from storage. Traditional models do not weigh the re-use of data in intensive computational processes in deciding whether to re-compute or store the data. In our project, we aim to calculate the computational and storage costs associated with conventional work-flows and generate a decision algorithm for computing or storing intermediate data. We plan to implement work-flows as Directed Acyclic Graphs (DAGs) with attributes for each node that reflect computational or storage costs. Our algorithm will then use this data to choose a core set of nodes based on factors such as computational cost, storage cost including I/O delay considerations, as well as introducing a factor which represents the frequency of reuse of individual nodes. This core-set will then represent the data that we need to store. We believe that using a balanced approach to combine these different factors will result in a more effective method for improving the efficiency of workflows in the cloud.

3. Theoretical Basis and Literature Review

3.1 Definition of the Problem

The standard programming paradigm thus far has been one of calculate, store, and retrieve. However, the rise of cloud computing, which readily provides for flexible allocation of computing resources,

renders this paradigm inefficient. This is because storing large volumes of rarely used data wastes space and energy – this in turn makes the standard paradigm very expensive. While computation and storage are equivalent, finding the balance between the two to maximize efficiency is very difficult.

3.2 Theoretical Background of the Problem

In the simple model of traditional computing, storage is used to hold the results of computations. In this system, results are simply re-called from storage when they are needed, and all results can be stored indefinitely. Cloud computing casts doubt onto this traditional computing model, as cloud computing offers readily available, flexibly allocated computational results. Instead of simply storing each and every result, it may be more efficient to simply store the provenance data and inputs to a process and recalculate results when necessary. This especially holds true for large datasets that are rarely used. The decision between storage and computation may seem simple at first, but there are many aspects to take into account to build a feasible and correct cost-benefit model. First, we must examine what can actually be stored and what can be feasibly computed. Is the goal to re-compute results exactly or merely find reasonable approximations of past results? Are there legal requirements or security issues? Second, a system that aims to enable re-computation will need additional metadata and provenance facilities in order to ensure that re-computation
Methods are known, and that result regenerations are successful. Third, the factors that determine when it is more efficient to re-compute a result, as opposed to storing it must be understood. These include factors such as the likelihood of reuse and the potential penalties if the data is unavailable when needed. Because this includes both user and system knowledge, understanding these factors is an important step in ensuring that the decision maker has all of the information required to make an informed decision. The main essence of the problem, however, can be boiled down to two simple costs –

The cost of storage (Cs) = cost per bit x bits stored

The cost of re- computation (Cr) = computation cost + miss penalty) x likelihood of reuse.

Consider an organization that needs Amazon's AWS cloud service as an example, and needs to access 5TB of data over a period of 10 years. Assume that if the data is unavailable when needed, the organization will lose $20,000. Also assume that the cost for storing the 5GB over 10 years will cost $90,000 and regenerating the data requires 5 days and 100 machines costing a total of $5,000. With these numbers alone it is unclear whether it is better to store persistently, or re-compute at a later date. With re computation, even a 50% chance per year of reuse may yield net savings provided there is sufficient lead time to regenerate results, while a lower chance of reuse may still dictate to store persistently if there is insufficient time to regenerate the data before it is needed. Therefore, much more work is needed to provide insight as to when the tradeoff should be taken, and if so, a model must be developed to determine what data should be stored in such a system.

3.3 Related Research to Solve the Problem

In a paper titled "Automating Analysis of the Computation-Storage Tradeoff", the authors examined how to model scientific workloads and associate those workloads with storage and computation costs. This was done in order to abstract storage and computation costs into discrete units which would aid in creating algorithms to determine which data sets to store. The authors decided to model scientific

workflows as a DAG's (Directed Acyclic Graph) in which individual nodes represented data sets and each edge represented a process which served to generate a node (or data set). A node that has no ancestors is an initial input, as it is not a result of any process within the DAG. A final result node is a node that has no children.

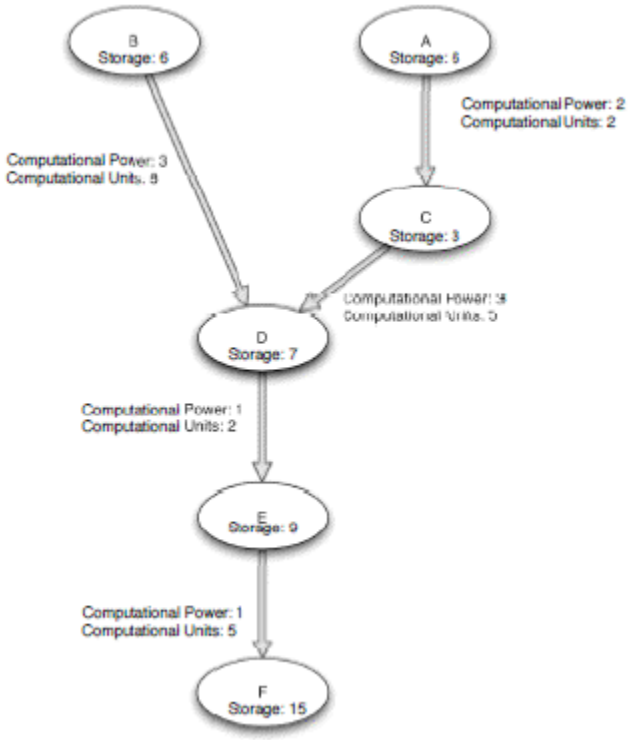


Figure 3.3.a: Example DAG

In the example DAG shown in Figure X, it can be seen that each node (or data set) has an associated storage cost, and that each edge (or process) has associated computation power and unit costs. The node storage cost is an abstracted generic storage unit, whereas computational power is the amount of computation that can be done in one time unit, and the computational units represent the amount of computation done. Thus, the amount of time an individual process takes is simply the computational units divided by the computational power.

The implementation consisted of two algorithms, one algorithm to select a core set of nodes to store, and a second verification algorithm to ensure that the nodes in the core set were sufficient to ensure that results could be re-calculated. If the verification algorithm finds that the final result cannot be recomputed from the nodes that are in the core set, the verification algorithm will add the necessary nodes to the core set. This second set of nodes is called a secondary set, and the final set is the combination of the core and secondary sets.

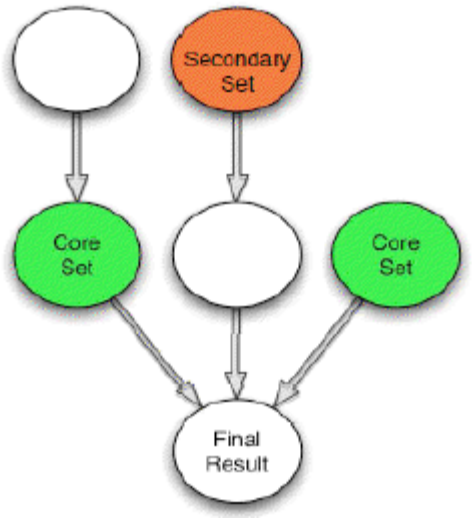


Figure 3.3.b: Core set nodes in (green) and Secondary set node in (orange)

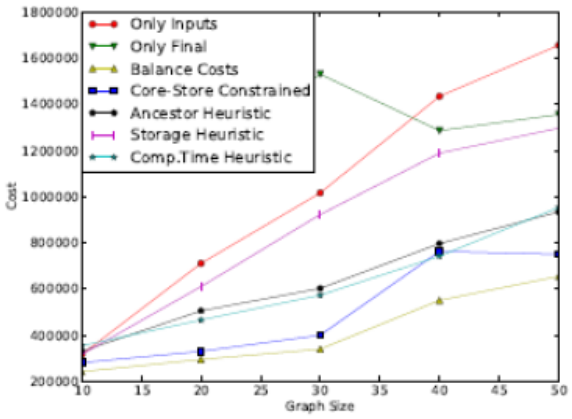
Five different algorithms were designed to select the core set, ranging from random selection routines to employing simple heuristics. The full list and details of each algorithm are shown in Table 3.1

Algorithm Name	Description
Randomized Cost Balance Search	This algorithm selects a random subset of nodes for the core-set. The algorithm runs for a user defined number of iterations and then picks the set of nodes with the lowest overall cost. This algorithm is intended to provide a good overall yearly cost in a simple fashion.
Randomized Constrained Core-Set	This algorithm selects a random core-set, limited by user specification, to try to reduce the overall amount stored by the system. Each core-set member is selected to be computationally expensive to reproduce so as to attempt to reduce the overall computation time and cost to reach the final result.
Greedy Heuristic: Number of Ancestors	This simple heuristic search sorts the nodes by the number of ancestors (parents) they have within the workflow DAG. It then iteratively selects the nodes with the most ancestors for the core-set and verifies re-computability like the previous algorithms. The number of selected nodes is tunable (the authors chose 15%).
Greedy Heuristic: Storage Amount	This algorithm sorts the nodes by the total amount of storage they use, and those with the lowest storage requirements are chosen for the core-set. This algorithm is purely for comparison purposes to see if reducing the size of the core-sets storage can improve total costs.
Greedy Heuristic: Computation Requirements	This particular algorithm sorts nodes by the total computational units that go into each given node. Nodes with the highest computational cost to produce are put into the core-set first. This algorithm is used to see if a simple-heuristic approach can impact the computational cost and time of regenerating a final result.

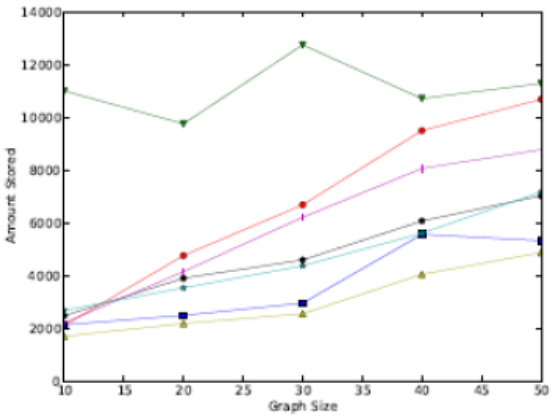
Table 3.3.c Core set selection algorithms

The algorithms shown in Table 3.3c were run on two different DAG generation schemes. The first, called random DAG generation, generates a random DAG with each node having one to three children. Nodes are randomly assigned storage costs, and likewise, edges are randomly assigned computational power and computational units. Costs are represented as a per-unit month of storage, and per unit of computation. Storage costs are calculated by adding the amount of storage needed multiplied by the amount of time the data set is stored. Computation costs are calculated simply by adding all the costs of each process' computation. The second type of DAG generation, called increasing DAG, has the amount of storage used by each node generally increasing as the nodes approach the final result. The computational power and units needed to generate a node are computed as a function of the amount stored.

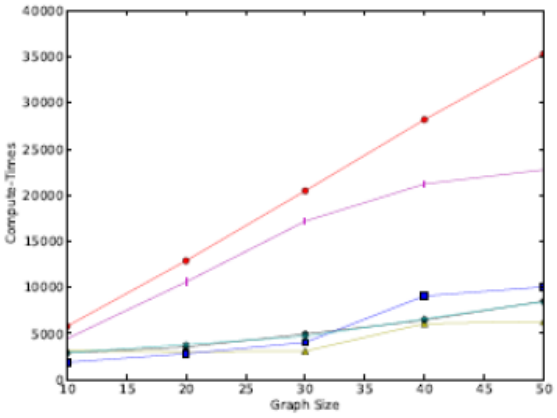
Each core selection algorithm was run 10 times for each graph size, and the results were averaged. Four different metrics were examined for evaluation – Total Cost: the combined cost to reach the final result given the storing data sets for twelve months, Total Storage Needed: the amount of storage needed to store the results for 12 months, Total Computation Time: total computation time needed across all edges (or processes), and Critical Path Time: the minimum amount of time needed to recomputed a result assuming as many processes as necessary can be run in parallel (provided the relevant input data is available).



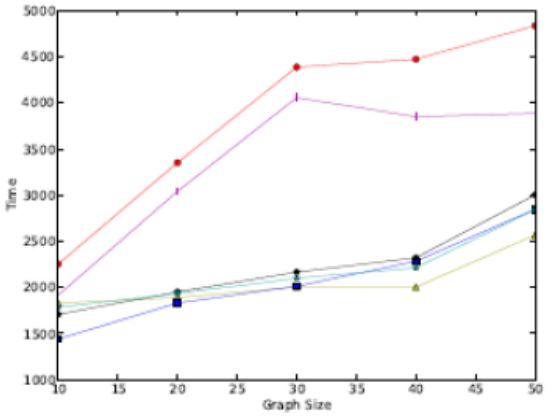
(a) Total Costs



(b) Storage Amount



(c) Total Computation Time Used



(d) Critical Path Time

Figure 3.3.d Results of simulation with random DAG generation

As can be seen in Figure 3.3, choosing to store intermediate datasets simultaneously yields improvements in total re-computation (critical path) time, storage amount, total costs, and computational time used. The graphs indicate a savings of up to approximately 20-60% over all methods when compared to the cost of storing just the inputs or final results. The storage based heuristic performed the worst, while the randomized cost balance search performed the best. This is because the storage algorithm attempts to minimize the storage amount when choosing the core set, however this means that the verification algorithm has to add more nodes into the final set, which creates a bigger overall storage amount and re-computation overhead. However, the results overall do indicate that it is possible to improve on computational and storage costs over simply storing the inputs, and that one approach is not guaranteed to give good results across all metrics.

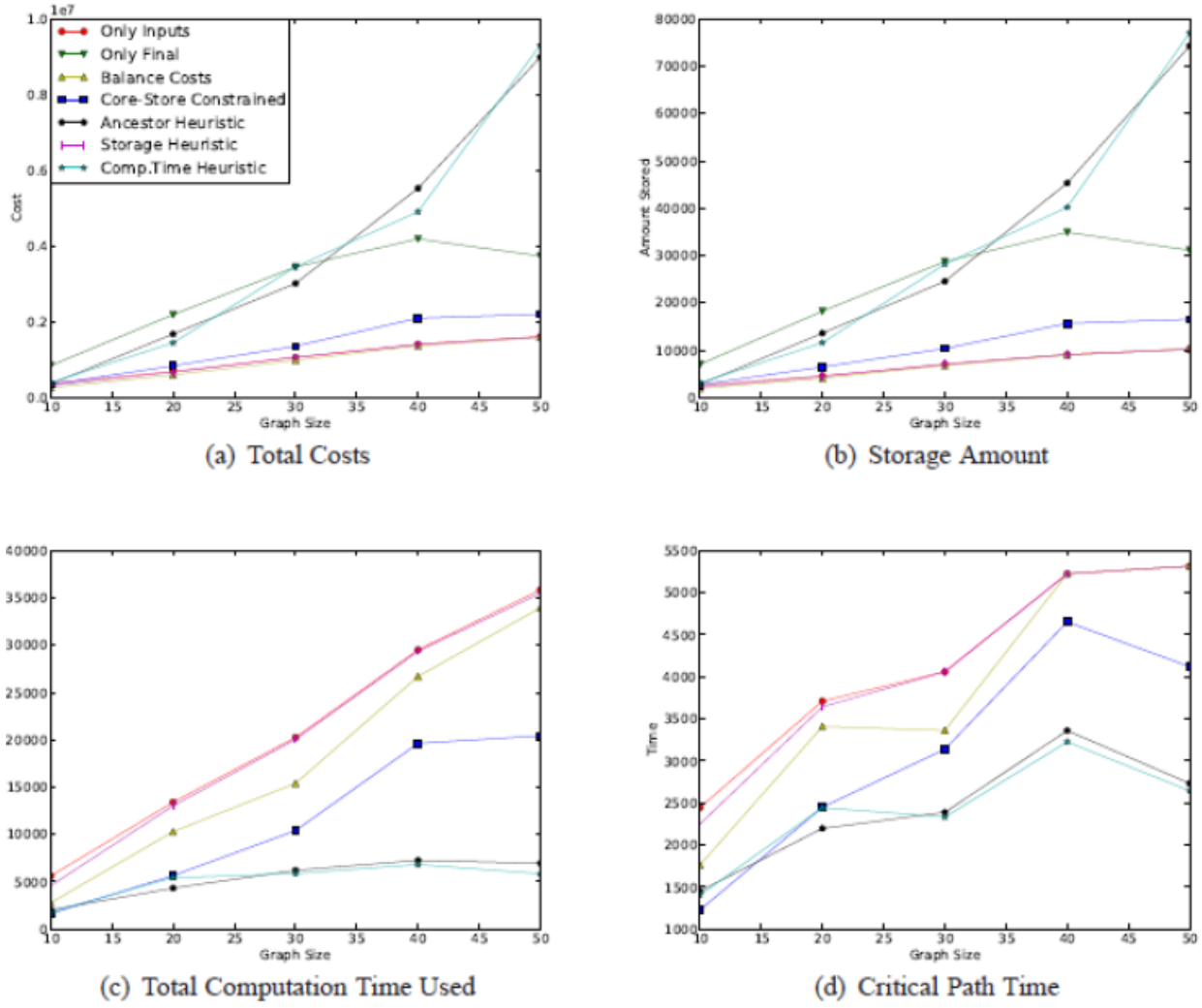


Figure 3.3.e: Results of simulation with increasing DAG generation

From Figure 3.3e, it can be seen that the randomized balanced cost and storage algorithms outperform the other algorithms in terms of total costs and storage amount. This is because the balanced cost seeks to store nodes with a good balance between storage and computation costs, and the storage algorithm is concerned with minimizing the storage cost. However, these algorithms fared much worse in terms of total computational time used and the critical path time. These results show that what works well for optimizing over one dimension can perform very poorly over another.

3.4 Advantage/Disadvantage of the Related Research

The main advantage of the research done in "Automating Analysis of the Computation-Storage Tradeoff" is that it shows the possibility to significantly impact the total costs and re-computation time associated with re-computing a final result. The research also shows the impact of different DAG's on the storage vs. computation tradeoff – what works best for one DAG does not necessarily mean it will be a good solution for another. The disadvantage of this research was that the authors were unable to find a good heuristic that would perform equally well under both DAG generation cases.

3.5 Our Solution to the Problem

We propose to add another attribute to each node of the DAG called the reuse factor. This number will be used to represent how frequently a node (or data set) is used – the higher the number, the more frequently the node is used. We believe this will help to make a more generic algorithm (resistant to DAG changes) because one of the motivations for trading storage vs. computation is that less frequently used datasets should not be stored. The authors of "Automating Analysis of the Computation-Storage Tradeoff" also failed to take into account the I/O penalties associated with fetching intermediate data sets for computation, and they did not include a real-world cost (in terms of dollars) for the total cost benchmark. By combining the cost of storage (in dollars) along with computational costs and factoring in the frequency of reuse of a node, we believe that we will be able to find a core-set selection algorithm that will perform equally as well on varying DAG's.

4. Hypothesis

Implementing an algorithm that can determine when the cost-storage tradeoff should be taken will enable a user to select a cost effective method of managing large datasets in cloud-based storage systems.

5. Methodology

5.1 Input Generation

Input generation is an integral part of this project, as we have proposed to run our algorithms on DAGs which are modeled after scientific workflows. At first, a random DAG generation scheme was to be implemented; however this idea was ultimately not used because a real-world data set would better serve the purpose of proving our algorithm's cost effectiveness. Thus, we identified two actual

workflows and manually translated the workflows into DAGs. The actual implementation and format of the DAGs are discussed in more detail in part 6, Implementation.

5.2 Problem Solution Design

The translated workflows are stored into XML files as DAGs, with nodes supporting annotatable attributes. Each node in the DAG represents a process input, output, or both, and also contains information on the process that generated it. Each node has a set of attributes, with the main parameters being: ***storage*** or the amount of storage space the node takes, ***computational power*** or the amount of computation that can be done in one time unit, ***computational units*** or the amount of computational necessary to produce the node, and ***reuse*** or the amount of times the node is reused. An example of the DAG in XML format can be found below in part 6, Implementation.

Two algorithms were designed to run on these translated workflows, with the aim of proving that even a simple algorithm can produce a cost-effective method of managing large datasets in the cloud. Table 5.2b further details the two algorithms. All monetary costs are taken from the Amazon Cloud Service, with dollar costs indicating costs per month as shown in Table 5.2a.

Cost Type	Dollar Amount
Computation Cost	$0.085 per hour
Data Transfer (I/O) Cost	$0.12 per GB per month
Storage Cost	10TB/ month at $0.12 per GB per month

Table 5.2.a: Amazon Cloud Service Monthly Costs

Algorithm Name	Description
Single Node	The algorithm calculates the cost of storage (C_S) and cost of re-computation (C_R) for each node in the DAG and decides to label a node for storage the node C_S < node C_R. If node C_S > node C_R, the algorithm will choose to label the node for re-computation instead. The cost formulas are as follows: C_S = (data size in GB x data transfer I/O cost) + (storage cost per GB/month x size of data in GB) C_R = reuse x [(data size in GB x data transfer I/O cost) + (computation cost per hour x re-computation time in hours)]
Multi Node	The multi node algorithm aims to include the real time costs of maintaining a multiple node workflow in the cloud. It takes into consideration that the re-computation cost of a particular node will include the re-computation costs of all its parent nodes that are not stored. Consider C_S is the cost of storage and C_R is the cost of re-computation. CR is the cost rate of a particular node with units of dollars per hour and this will be set to C_S or C_R depending on the flag on the node. The algorithm first scans all the input nodes and flags them for storing. Then it advances through its child nodes and calculates the cost of

	maintaining each node. The cost of storing a particular node is the same as before with the single node algorithm. C_S = (data size in GB x data transfer I/O cost) + (storage cost per GB/month x size of data in GB) However, in order to calculate the cost of recomputation, we first check the reuse factor. If the reuse factor is greater than zero, we calculate the sum of all the recomputation costs for all the parent nodes that have not been flagged for storage up to the first node that is stored. We then add this total to the cost of recomputation of the current node from its immediate parents. The formula for cost of recomputation is again the same as before, except we apply it on a much larger scale. We decided to not include the I/O cost with the multi node algorithm since the intermediate node infomration is mostly shuttled around within the cloud and not downloaded which is what incurs the I/O cost. If the reuse factor is 0, we ensure that all of its child nodes either have reuse = 0, or are flagged for storage. Then we consider the recomputation cost of only the current node and use that as $C_{R.}$ C_R = reuse x [(computation cost per hour x re-computation time in hours)] + total of all parent C_R Finally, CR = $C_S < C_R$? C_S : C_R

Table 5.2.b: Algorithm Design Description Table

5.3 Output Generation & Hypothesis Checking

Our entire solution is implemented in python; a XML parser was first written to read the DAG into an object in python, after which the two decision algorithms would be run. The decision algorithms report the storage cost and re-calculation cost of each node, along with indicating whether the node has been labeled for storage or re-computation (non-storage). In order to test against our hypothesis, the decision algorithms will also report the total storage cost of all nodes labeled for storage, the total re-computation cost for all nodes labeled for re-computation, and the total storage cost for simply storing all nodes. Thus, the hypothesis may be proved if the total cost of storage for storing all nodes is greater than the sum of the total re-computation and total storage costs of the decision algorithm, or disproved if the opposite is true.

6. Implementation

6.1 XML File for Single node

As mentioned in part 5 above, we identified two real-world workflows and manually translated the workflows into DAGs. The DAGs are recorded in a XML file with a sample format as follows:

```
<?xml version="1.0" ?>
<DAG>
```

```
  <nodes>
    <node>
      <ID>1</ID>
      <storage>5</storage>
      <reuse>1</reuse>
      <level>0</level>
      <compPower>2</compPower>
      <compUnits>1800</compUnits>
      <child>2</child>
      <parents>
        <parent>0</parent>
      </parents>
    </node>

    <node>
      <ID>2</ID>
      <storage>5</storage>
      <reuse>1</reuse>
      <level>1</level>
      <compPower>2</compPower>
      <compUnits>1800</compUnits>
      <child>3</child>
      <parents>
        <parent>1</parent>
      </parents>
    </node>

    <node>
      <ID>3</ID>
      <storage>5</storage>
      <reuse>1</reuse>
      <level>2</level>
      <compPower>2</compPower>
      <compUnits>1800</compUnits>
      <child>4</child>
      <parents>
        <parent>2</parent>
      </parents>
    </node>
  </nodes>
</DAG>
```

Listing 6.1.a: Sample DAG Implementation in a XML File

Each individual data set in the workflow is treated as a node with eight attributes, which are each assigned a corresponding value.

XML file for Multi Node

```
<?xml version="1.0" ?>
```

```
<DAG>
<nodes>
    <node>
      <ID>1</ID>
      <storage>0</storage>
      <reuse>1</reuse>
      <compUnits>1800</compUnits>
      <child>2</child>
      <parents>
        <parent></parent>
      </parents>
      <flag>1</flag>
    </node>

    <node>
      <ID>2</ID>
      <storage>20</storage>
      <reuse>0.00416666667</reuse>
      <compUnits>0.45</compUnits>
      <child>3</child>
      <parents>
        <parent>1</parent>
      </parents>
      <flag>1</flag>
    </node>
```

```
<node>
   <ID>3</ID>
   <storage>90</storage>
   <reuse>0.00416666667</reuse>
   <compUnits>13.1666667</compUnits>
   <child>4</child>
   <parents>
      <parent>2</parent>
   </parents>
   <flag>1</flag>
</node>

<node>
   <ID>4</ID>
   <storage>90</storage>
   <reuse>0.0104166667</reuse>
   <compUnits>5</compUnits>
   <child>5</child>
   <parents>
      <parent>3</parent>
   </parents>
   <flag>1</flag>
</node>
```

```
<node>
  <ID>5</ID>
  <storage>0.016</storage>
  <reuse>0.00416666667</reuse>
  <compUnits>1.33333333</compUnits>
  <child>6</child>
  <parents>
    <parent>4</parent>
  </parents>
  <flag>1</flag>
</node>

<node>
  <ID>6</ID>
  <storage>0.000001</storage>
  <reuse>0.00416666667</reuse>
  <compUnits>0.0166666667</compUnits>
  <child>7</child>
  <parents>
    <parent>5</parent>
  </parents>
  <flag>1</flag>
</node>

<node>
```

```
        <ID>7</ID>
        <storage>0.000025</storage>
        <reuse>0.00416666667</reuse>
        <compUnits>4.08333333</compUnits>
        <child></child>
        <parents>
          <parent>6</parent>
        </parents>
        <flag>1</flag>
    </node>
  </nodes>
</DAG>
```

6.2 Workflow Used for the 'Single Node' Algorithm and Flowchart

The first example workflow is from http://www.usenix.org/event/hotcloud09/tech/slides/adams.pdf, and is an archive of 100,000 pictures in five different formats (bmp, jpeg, tiff, gif, and png) managed with Amazon Web Services. In this example, the entire archive requires 2.2 TB of space. In this case, the raw bitmap is the sole input of the DAG, with all other four formats deriving from this input node. Another important point to mention is that there are no intermediate data sets in this workflow. Two test cases were used for this example workflow: 1) All nodes with reuse = 1, and 2) Mixed reuse. Figure X shows an illustration of the DAG derived from this example workflow, and Tables X and X provide a summary of the node attribute values for each case. Listing X shows a quick flowchart of the 'Single Node' Algorithm (please see Listing X in part 10, appendices for the complete python code). Please see Listing X in part 10 appendices, for a complete listing of the actual XML files used.

Case	Node Attribute				
Reuse = 1	ID	Reuse	Storage (GB)	Comp. Power	Comp. Units
	1	1	1024	1	1
	2	1	150	2	2
	3	1	350	2	4
	4	1	500	2	6
	5	1	228.8	2	3

Table 6.2.a: Node Attributes for Case: All Reuse = 1

Case	Node Attribute				
Mixed Reuse	ID	Reuse	Storage (GB)	Comp. Power	Comp. Units
	1	1	1024	1	1
	2	2	150	2	2
	3	1	350	2	4
	4	2	500	2	6
	5	1	228.8	2	3

Table 6.2.b : Node Attributes for Case: Mixed Reuse

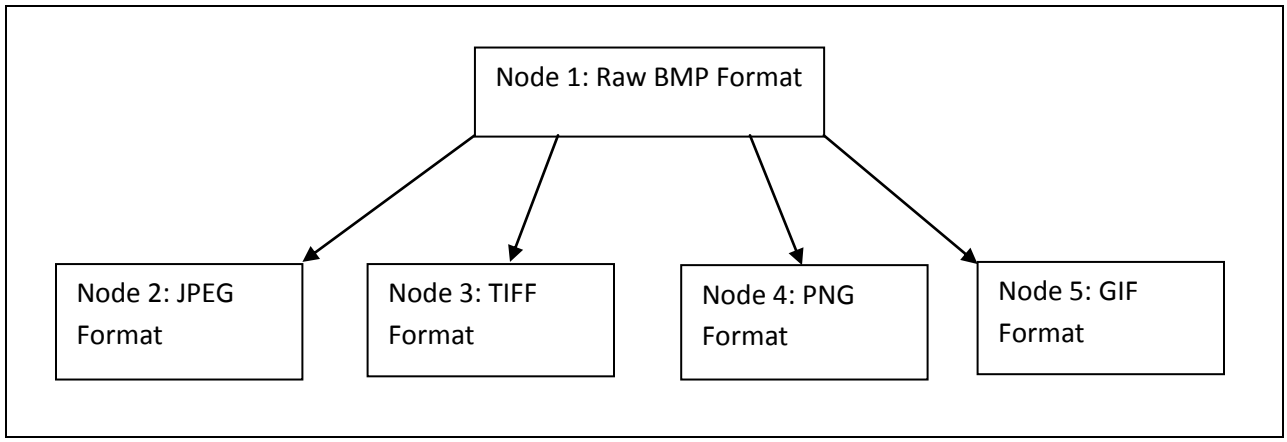
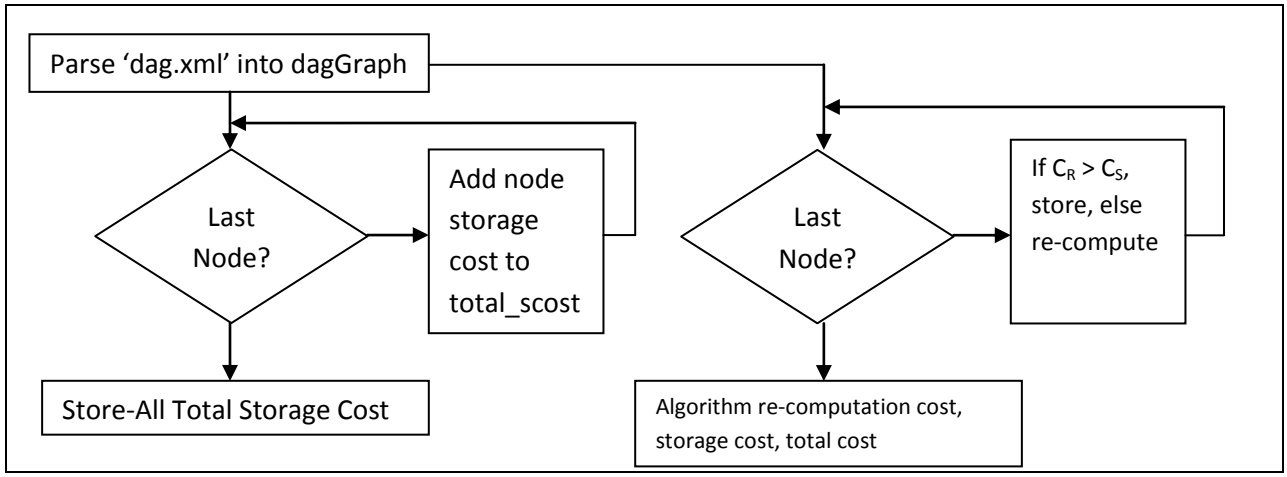


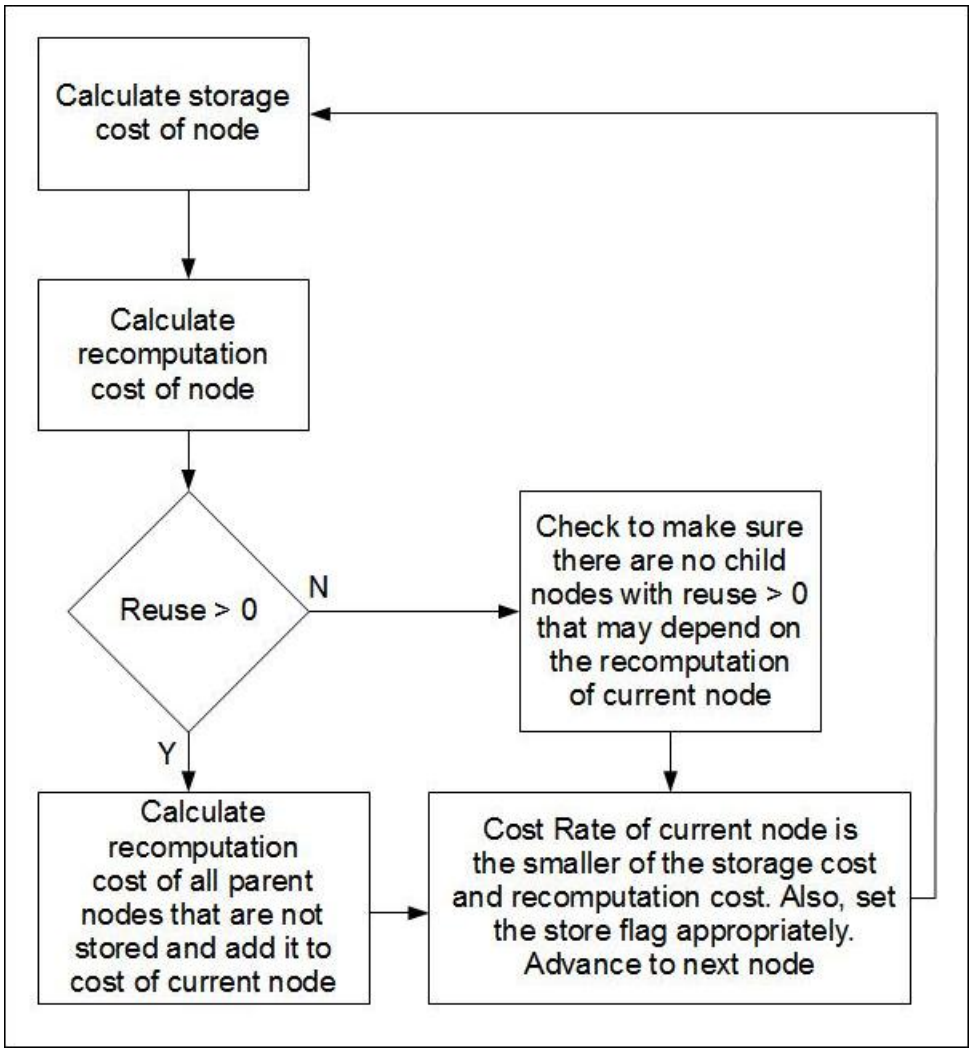
Figure 6.2.c: DAG for Picture Collection Example Workflow



Listing 6.2.d: 'Single Node' Algorithm Flowchart

Workflow Used for the 'Multi Node' Algorithm and Flowchart

We used the workflow from "A Cost-Effective Strategy for Intermediate Data Storage in Scientific Cloud Workflow Systems" by Dong Yuan, Yun Yang, Xiao Liu, Jinjun Chen. As you can see from the following image, there are multiple intermediate nodes and all of them had a significant reuse rate. The De-dispersion files were used once every 4 days and all the other nodes were reused once every 10 days. This was accurately represented in the input



7. Data Analysis and Discussion

7.1 Results and Discussion for the 'Single Node' Algorithm

Table 7.1 show the results (store or not store) of the 'Single Node' Algorithm run on the picture archive example described in part 6 above, and Table 7.1 compares the costs of the 'Single Node' Algorithm against a 'Store-All' method for the same workflow.

ID	'Single Node' Algorithm Result					
	Case: All Reuse = 1			Case: Mixed Reuse		
	Node Storage Cost	Node Re-computation Cost	Result	Node Storage Cost	Node Re-computation Cost	Result
1	225.28	122.965	Store	225.28	122.965	Store
2	33.0	18.085	Not Store	33.0	36.17	Store

3	77.0	42.17	Not Store	77.0	42.17	Not Store
4	110.0	60.255	Not Store	110.0	120.51	Store
5	50.336	27.5835	Not Store	50.336	27.5835	Not Store

Table 7.1: 'Single Node' Algorithm Results Table

Case	Store-All Total Storage Cost	'Single Node' Total Re-computation Cost	'Single Node' Total Storage Cost	'Single Node' Total Cost
All Reuse = 1	495.616	148.0935	225.28	373.3735
Mixed Reuse	495.616	69.7535	368.28	438.0335

Table 7.2: Comparison of 'Store-All' vs. 'Single Node' Algorithm

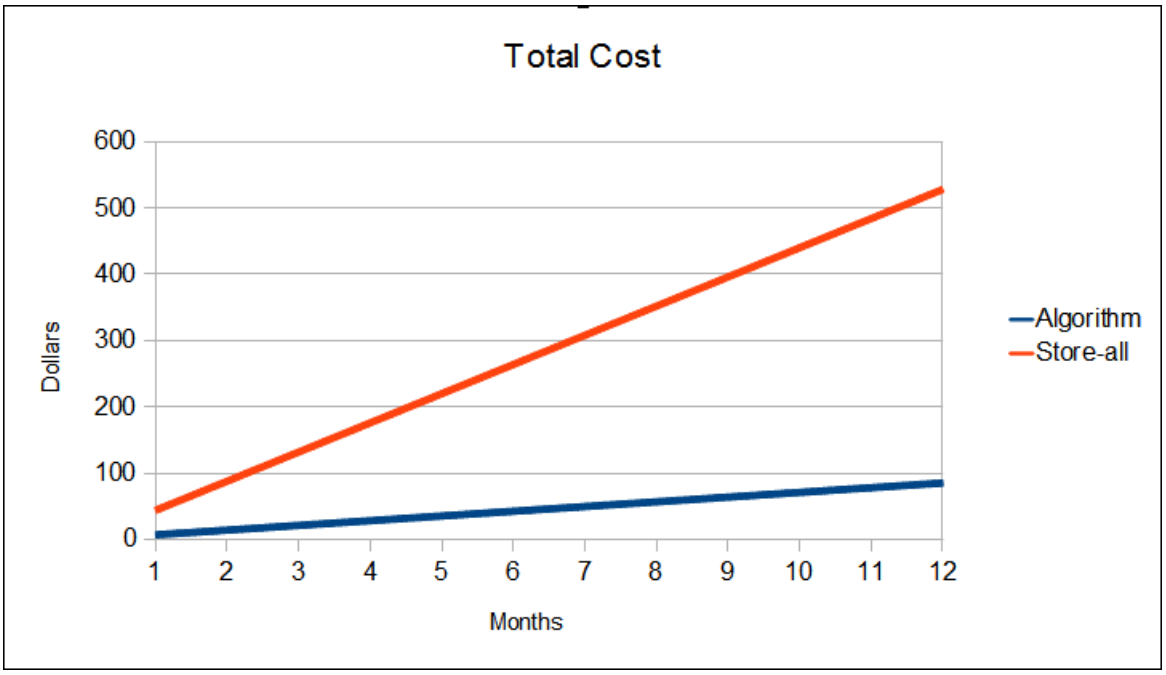
The results in Table 7.2 (Comparison of 'Store-All' vs. 'Single Node') shows that even with a simple decision method, it is possible to reduce the cost involved with managing a large dataset in the cloud. In the case of All Reuse = 1, there is a cost savings of $122.2425 per month; the cost savings for the mixed reuse case is $57.5825 per month. This validates the hypothesis stated in part 4, which states that even a simple decision algorithm can produce a cost-effective solution for dataset management in cloud-based storage systems.

Upon inspection of Table 7.2 ('Single Node' Algorithm Results), it is readily observable that the algorithm is highly dependent upon the reuse factor. The All Reuse = 1 case yields the result that all nodes (besides node 1 which is the input) will not be stored. On the other hand, the Mixed Reuse case shows that nodes with a reuse of higher than 1 will be stored when the other costs such as storage and computation costs remain the same. This is because our re-computation cost formula, C_R = reuse x [(data size in GB x data transfer I/O cost) + (computation cost per hour x re-computation time in hours)], is highly sensitive to the reuse factor; a reuse factor of two effectively doubles the original cost of re-computation, making this 'Single Node' algorithm far from an optimal tradeoff analyzer.

The results of the 'Single Node' algorithm prove that our hypothesis is correct; indeed it is possible to affect the cost of data set management with an algorithm that analyzes the associated costs. However, the sensitivity of the 'Single Node' algorithm motivated the pursuit of a second algorithm which would mitigate not only the effects of the reuse factor, but to compare the access time costs as well.

Results and Discussion for the 'Multi Node' Algorithm

As you can see from the image below that the multi node algorithm provides a significant cost advantage over the store-all methodology. The algorithm pegs the cost of maintaining this workflow in the cloud at a little over $7 per month while the store-all methodology costs $44 per month.



These were the results for each node:

Node ID: 1 Flag: 1 Cost for a month: 0.0

Node ID: 2 Flag: 0 Cost for a month: 0.11642025009313624

Node ID: 3 Flag: 0 Cost for a month: 3.406411521298537

Node ID: 4 Flag: 0 Cost for a month: 3.2369119372259347

Node ID: 5 Flag: 0 Cost for a month: 0.34609529461449384

Node ID: 6 Flag: 1 Cost for a month: 0.00087757599397812

Node ID: 7 Flag: 1 Cost for a month: 0.002193939984945306

8. Conclusions and recommendations

8.1 Conclusion

From above we can see that when reuse factor is 1, you will find the re-computation cost low and when we take mixed reuse factors of 1 and 2, we could find some nodes to store and some nodes to be recomputed.

Finally from this results we can say that re-use factor has very important role in both Algorithms to calculate the values for re-computation

If you do re-use intermediate nodes independently, there is an extraneous use for that node other than for the workflow itself and this adds to the cost of computation

However, for the nodes whose only purpose is to compute the initial set of stored nodes, we apply the strategy of the single node algorithm to yield us the most economic option.

8.2 Recommendation for future studies:

Our current work focuses narrowly on a single aspect of the store vs. re-compute tradeoff mainly focusing with reuse factor. Even with our simple model and simulation there are a great number of other variables, algorithms and heuristics to examine. These include simulated annealing techniques, varying computational power, shifting prices across multiple dimensions, adding in hard time or storage constraints etc.

We'd also like to continue examining some of the more nebulous aspects of the store vs. re-compute tradeoff such as data unavailability penalty and delayed return penalty which plays very important role to find the re-computation cost to compare with storage cost.

9 Bibliography

Adams, D. D. E. Long, E. L. Miller, S. Pasupathy, and M. W. Storer, "Maximizing Efficiency By Trading Storage for Computation," in Workshop on Hot Topics in Cloud Computing (HotCloud'09), pp. 1-5, 200

Adams, Ian F., and Biran A. Madden. *Automating Analysis of the Computation-Storage Tradeoff*. Thesis. UC Santa Cruz, 2010. Web. 06 Nov. 2011. <http://www.ict.swin.edu.au/personal/dyuan/doc/IPDPS_DataStorage.pdf>.

Dong Yuan; Yun Yang; Xiao Liu; Jinjun Chen; , "A cost-effective strategy for intermediate data storage in scientific cloud workflow systems," *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* , vol., no., pp.1-12, 19-23 April 2010
URL:http://ieeexplore.ieee.org/stamp/stamp.jsp?p=&arnumber=5470453&isnumber=5470342

10. Appendix

10.1 DAG XML File Listing

Listing 10.1: DAG XML File for Picture Archive Case: All Reuse = 1

```
<?xml version="1.0" ?>
<DAG>
  <nodes>
    <node>
      <ID>1</ID>
      <storage>1024</storage>
```

```
    <reuse>1</reuse>
    <level>0</level>
    <compPower>1</compPower>
    <compUnits>1</compUnits>
    <child>2</child>
    <parents>
      <parent>0</parent>
    </parents>
  </node>

  <node>
    <ID>2</ID>
    <storage>150</storage>
    <reuse>1</reuse>
    <level>1</level>
    <compPower>2</compPower>
    <compUnits>2</compUnits>
    <child>0</child>
    <parents>
      <parent>1</parent>
    </parents>
  </node>

  <node>
    <ID>3</ID>
    <storage>350</storage>
    <reuse>1</reuse>
    <level>2</level>
    <compPower>2</compPower>
    <compUnits>4</compUnits>
    <child>0</child>
    <parents>
      <parent>1</parent>
    </parents>
  </node>

  <node>
    <ID>4</ID>
    <storage>500</storage>
    <reuse>1</reuse>
    <level>3</level>
    <compPower>2</compPower>
    <compUnits>6</compUnits>
    <child>0</child>
    <parents>
      <parent>1</parent>
    </parents>
  </node>
```

```
                    <node>
        <ID>5</ID>
        <storage>228.8</storage>
        <reuse>1</reuse>
        <level>3</level>
        <compPower>2</compPower>
        <compUnits>3</compUnits>
        <child>0</child>
        <parents>
          <parent>1</parent>
        </parents>
      </node>

   </nodes>
</DAG>
```

Listing 10.2: DAG XML File for Picture Archive Case: Mixed Reuse

```
<?xml version="1.0" ?>
<DAG>
   <nodes>
      <node>
        <ID>1</ID>
        <storage>1024</storage>
        <reuse>1</reuse>
        <level>0</level>
        <compPower>1</compPower>
        <compUnits>1</compUnits>
        <child>2</child>
        <parents>
          <parent>0</parent>
        </parents>
      </node>

      <node>
        <ID>2</ID>
        <storage>150</storage>
        <reuse>2</reuse>
        <level>1</level>
        <compPower>2</compPower>
        <compUnits>2</compUnits>
        <child>0</child>
        <parents>
          <parent>1</parent>
        </parents>
      </node>
```

```
    <node>
      <ID>3</ID>
      <storage>350</storage>
      <reuse>1</reuse>
      <level>2</level>
      <compPower>2</compPower>
      <compUnits>4</compUnits>
      <child>0</child>
      <parents>
        <parent>1</parent>
      </parents>
    </node>

    <node>
      <ID>4</ID>
      <storage>500</storage>
      <reuse>2</reuse>
      <level>3</level>
      <compPower>2</compPower>
      <compUnits>6</compUnits>
      <child>0</child>
      <parents>
        <parent>1</parent>
      </parents>
    </node>

                <node>
      <ID>5</ID>
      <storage>228.8</storage>
      <reuse>1</reuse>
      <level>3</level>
      <compPower>2</compPower>
      <compUnits>3</compUnits>
      <child>0</child>
      <parents>
        <parent>1</parent>
      </parents>
    </node>

  </nodes>
</DAG>
```

Listing 10.3: 'Single Node' Algorithm Output for Picture Archive Case: All Reuse = 1

```
calculating costs for store-all
```

```
node ID:  1 node storage cost:  225.28
node ID:  3 node storage cost:  77.0
node ID:  2 node storage cost:  33.0
node ID:  5 node storage cost:  50.336
node ID:  4 node storage cost:  110.0
store-all total storage cost (in $):  495.616

calculating costs for storage algorithm

node ID:  1 node recomp cost:  122.965
node ID:  1 node storage cost:  225.28 ---> STORED

node ID:  3 node recomp cost:  42.17
node ID:  3 node storage cost:  77.0 ---> NOT STORED

node ID:  2 node recomp cost:  18.085
node ID:  2 node storage cost:  33.0 ---> NOT STORED

node ID:  5 node recomp cost:  27.5835
node ID:  5 node storage cost:  50.336 ---> NOT STORED

node ID:  4 node recomp cost:  60.255
node ID:  4 node storage cost:  110.0 ---> NOT STORED

storage algorithm total re-computation cost (in $):  148.0935
storage algorithm total storage cost (in $):  225.28
storage algorithm total cost (in $):  373.3735
```

Listing 10.4: 'Single Node' Algorithm Output for Picture Archive Case: Mixed Reuse

```
calculating costs for store-all

node ID:  1 node storage cost:  225.28
node ID:  3 node storage cost:  77.0
node ID:  2 node storage cost:  33.0
node ID:  5 node storage cost:  50.336
node ID:  4 node storage cost:  110.0
store-all total storage cost (in $):  495.616

calculating costs for storage algorithm

node ID:  1 node recomp cost:  122.965
node ID:  1 node storage cost:  225.28 ---> STORED

node ID:  3 node recomp cost:  42.17
node ID:  3 node storage cost:  77.0 ---> NOT STORED

node ID:  2 node recomp cost:  36.17
```

```
node ID:  2 node storage cost:  33.0 ---> STORED

node ID:  5 node recomp cost:  27.5835
node ID:  5 node storage cost:  50.336 ---> NOT STORED

node ID:  4 node recomp cost:  120.51
node ID:  4 node storage cost:  110.0 ---> STORED

storage algorithm total re-computation cost (in $):  69.7535
storage algorithm total storage cost (in $):  368.28
storage algorithm total cost (in $):  438.0335
```

10.2 Python Code Listing

Listing 10.5: DAG_Reader.py

```
import xml.dom.minidom

DAG_FILE="dag.xml"

class DAGParser(object):

   def __init__(self, file):
      self.DAG = {}
      self.doc = xml.dom.minidom.parse(DAG_FILE)
      self.dom = self.doc.documentElement
      self.parseDAG(self.dom)
      #print(self.DAG)

   def parseDAG(self, dom):
      # Reading from <nodes> <node></node> <node></node> ... </nodes>
      for node in dom.getElementsByTagName('nodes')[0].getElementsByTagName("node"):
         self.parseNode(node)

   def parseNode(self, node):
      ID = self.getText(node.getElementsByTagName("ID")[0].childNodes)
      storage = self.getText(node.getElementsByTagName("storage")[0].childNodes)
      reuse = self.getText(node.getElementsByTagName("reuse")[0].childNodes)
      level = self.getText(node.getElementsByTagName("level")[0].childNodes)
      compPower = self.getText(node.getElementsByTagName("compPower")[0].childNodes)
      compUnits = self.getText(node.getElementsByTagName("compUnits")[0].childNodes)
      next = self.getText(node.getElementsByTagName("child")[0].childNodes)
      prev = []
      # Reading from <parents> <parent></parent> <parent></parent> </parents>
      for el in node.getElementsByTagName("parents")[0].getElementsByTagName("parent"):
         prev.append(self.getText(el.childNodes))
      self.DAG[ID] = {'storage': storage, 'reuse': reuse, 'level': level,
                  'compPower': compPower, 'compUnits': compUnits,
```

```
                    'next': next, 'prev': prev}

    def getText(self, childNodes):
        rstring = ""
        for child in childNodes:
            if child.nodeType == child.TEXT_NODE:
                rstring = rstring + child.data
        return rstring


DAGParser(DAG_FILE)
```

Listing 10.6: Storage_ Algorithm.py ('Single Node' Algorithm)

```
""" Storage_Algorithm.py: uses DAG_Reader.py to parse the sample xml file
    containing a DAG and returns the cost of store-all and the storage and
    re-computation costs of the nodes selected by the storage algorithm
"""

from DAG_Reader import DAGParser

DAG_FILE="dag2.xml"

# storage/re-computation costs from Amazon Cloud
computation_cost = .085
io_cost = .12
storage_cost = .1

# formula to calculate storage costs
def calc_storage(storage, store_cost, io_cost):
    total_storage_cost = (storage * io_cost) + (storage * store_cost)
    return total_storage_cost

# formula to calculate re-computation costs
def calc_recomp(storage, reuse, store_cost, io_cost, comp_cost, comp_time):
    recomp_cost = reuse * ((storage * io_cost) + (comp_time * comp_cost))
    return recomp_cost

# create DAGParser object
dagGraph = DAGParser(DAG_FILE)

# find the storage cost for 'store-all'
print 'calculating costs for store-all\n'
total_scost = 0

for ID in dagGraph.DAG:
    node = dagGraph.DAG[ID]
    storage = float(node['storage'])
```

```
    scost_per_node = calc_storage(storage, storage_cost, io_cost)
    total_scost = total_scost + scost_per_node
    print 'node ID: ', ID, 'node storage cost: ', scost_per_node

print 'store-all total storage cost (in $): ', total_scost, '\n'

# storage algorithm:
""" determines the cost of storage and the cost of re-computation for all
    nodes in the DAG and stores based on the following procedure:

    if node re-computation cost > node storage cost ---> STORE
    if node re-computation cost < node storage cost ---> RE-COMPUTE (or NOT STORE)
"""

print 'calculating costs for storage algorithm\n'
total_rcost = 0
total_scost = 0

for ID in dagGraph.DAG:
    node = dagGraph.DAG[ID]
    reuse = float(node['reuse'])
    storage = float(node['storage'])
    comp_time = float(node['compUnits']) / float(node['compPower'])

    node_scost = calc_storage(storage, storage_cost, io_cost)
    node_rcost = calc_recomp(storage, reuse, storage_cost, io_cost,
                             computation_cost, comp_time)
    print 'node ID: ', ID, 'node recomp cost: ', node_rcost

    if node_scost > node_rcost and int(ID) != 1:
        # node should NOT be stored, so add node_rcost to total_rcost
        print 'node ID: ', ID, 'node storage cost: ', node_scost, '---> NOT STORED\n'
        total_rcost = total_rcost + node_rcost
    else:
        print 'node ID: ', ID, 'node storage cost: ', node_scost, '---> STORED\n'
        total_scost = total_scost + node_scost

print 'storage algorithm total re-computation cost (in $): ', total_rcost
print 'storage algorithm total storage cost (in $): ', total_scost
print 'storage algorithm total cost (in $): ', total_scost + total_rcost
```