Cost Optimization of Cloud-Based Data Integration System

Peng Zhang
Institute of Computing Technology, Chinese Academy of Sciences,
Graduate University of Chinese Academy of Sciences,
North China University of Technology,
Beijing, China
Email:zhangpeng@software.ict.ac.cn

Yanbo Han, Zhuofeng Zhao, Guiling Wang
North China University of Technology,
Beijing, China
Email:yhan@ict.ac.cn
Email:wangguiling@software.ict.ac.cn
Email:zhaozf@software.ict.ac.cn

***Abstract*—Cloud computing provides virtualized, dynamically-scalable computing power. At the same time, reduction of cost is also considered as an important advantage of cloud computing. Data integration can notably benefit from cloud computing because integrating data is usually an expensive task. However, existing optimization techniques pay less attention on the fact that different execution plans of the same data integration application generate different usage costs while cloud computing provides good enough performance, so this paper introduces the cost optimization of cloud-based data integration system. The data integration system's data service layer facilitates accessing and composing information from a range of enterprise data sources through data service composition. In addition, two task scheduling algorithms for parallel part and non-parallel part are proposed to minimize the usage cost required to complete the execution of composite data service when computational capability provided by cloud computing is charged. Both of the two can obtain optimal plans in polynomial time. Experiments with the system indicate that our algorithms can lead to significant cost saving over more straightforward techniques.**

Keywords-data service; cost optimization; cloud computing; data integration.

I. INTRODUCTION

Cloud adopts an on-demand infrastructure to provide its computing resource as an elastic service. Existing technologies highlight the benefits cloud users can gain. Cloud users can apply for resources from cloud and then return back to cope with their business needs. By the pricing model of paying-as-you-go, cloud users only pay for the resources allocated to them. This can benefit cloud users by transferring the risk of economic losses caused by over proportioning and under-provisioning to cloud providers[1]. Data Integration can notably benefit from cloud computing because accessing multiple data sources and integration of instance data are usually expensive tasks. For example, entity similarity computation usually has quadratic complexity and is thus very expensive for large-scale data integration[6].

However, when cloud computing provides good enough performance, the lack of approaches and tools to analyze the economic efficiency results in a weak foundation for the cost optimization. In this paper, we are interested in the more basic challenge of economic efficiency when cloud computing provides good enough performance. To this end we propose the cost optimization of cloud-based data integration system-DeCloud. The main contributions are as follows:

- We present the DeCloud architecture to facilitate accessing and composing information from a range of enterprise data sources through data service composition.
- We present two task scheduling algorithms for parallel part and non-parallel part of composite data service. The two algorithm help minimizes the usage cost required to complete the execution of composite data service when computational capability provided by cloud computing is charged. Both of the two can obtain optimal plans in polynomial time.
- We evaluate the algorithms and demonstrate that it can significantly achieve cost saving over more straightforward techniques.

The paper is organized as follows: Section 2 gives the system architecture. Section 3 gives an example to formulate the cost optimization of execution plan. Section 4 introduces the two cost optimization algorithms for execution plan, two detailed examples are introduced. Section 5 introduces the implement and experiments. Section 6 introduces related works. Section 7 sums up with several concluding remarks.

II. SYSTEM ARCHITECTURE

Extended from our previous Web Service Management System with Multiple Engines[2], the cloud-based data integration system-DeCloud is developed, it consists of three major components, see Figure 1. The Metadata component deals with metadata management, registration of new data services, and data service composition that generates integrated views provided to the client. The data service, which separates the data access interface from the underlying heterogeneous components in a way that facilitates their seamless composition, enables a virtual layer and provides a new foundation for building enterprise integration applications that need to access and compose information from a range of enterprise data sources, such as XML, HTML, Database, and Excel.

Given an integrated view of the data service composition, a client can request it through the Spreadsheet Client Interface, and then the Query Processing and Cost Optimization component handles optimization and execution of such request, i.e., it chooses and executes a minimum cost

978-0-7695-4819-7/12 $26.00 © 2012 IEEE
DOI 10.1109/WISA.2012.13

plan whose operators invoke the relevant composite data service which comprises several data services. And then the composite data service returns results from a range of enterprise data sources through data service layer.

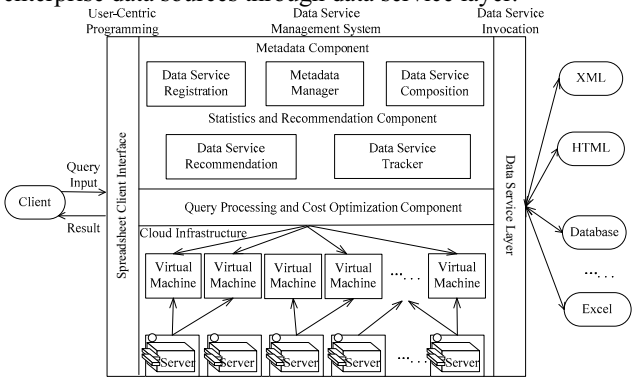


Figure 1. The DeCloud Architecture

The Profiling and Statistics component profiles data services for their response time, update frequencies, request frequencies, and maintains relevant statistics over the machines, to the extent possible. This component is used primarily by the performance optimization that achieves the best performance with minimum maintenance cost[4].

In some sense, our DeCloud architecture is similar to mediators in data integration system[5, 9, 14, 18, 19], what makes DeCloud different from them is the data service layer. In addition, cloud infrastructure is also an important feature. We argue that, multiple machines provided by cloud services could be exploited to achieve parallelism in execution and reduce the response time to the user. One can expect some reasonable benefits because of the following reasons:

- A single machine might be incapable to finish the whole process especially when computational capability and resources are insufficient in a separate machine.
- Cutting the composite data service into execution segments could achieve parallelism and load balance over multiple machines.
- Multiple machines could provide reliability guarantee in a certain degree.
- In cloud, a large number of third-party businesses make money out of cloud services. They either charge money on per service basis (micro money) or through advertising.

In DeCloud, operators of composite data services are executed within one or multiple operator-specific tasks that can be run in parallel on distributed machines. Tasks may concurrently store their results in a distributed data store. Furthermore, tasks can be executed as soon as computing resources are available in the cloud to obtain a minimal overall execution time. Finally, our execution model is based on a dynamic invocation of tasks such that each finished task invokes the generation of all operators following in the composite data service.

To support efficient, parallel execution of composite data service, we support both intra and inter-operator parallelism similar as in parallel database systems. In addition to pipeline parallelism between adjacent operators, data partitioning is utilized to run independent operators on different data in parallel and to parallelize operators on disjoint data partitions. However, intra-operator parallelism is subject to partitionable input data (i.e., operator parameters) only. A parameter p_k of an n-ary operator op is called partitionable if and only if the following two properties hold: (1) The parameter p_k is a set of tuples S. (2) For any complete and disjoint partitioning $p_k = p_{k1} \cup p_{k2}$ of p_k holds: $op(p_1,\ldots, p_k,\ldots,p_n) = op(p_1,\ldots,p_{k1},\ldots,p_n) \cup op(p_1,\ldots,p_{k2},\ldots,p_n)$. We call an operator blocking if none of its parameters is partitionable. A blocking operator is therefore unable to produce any (partial) result until all input data is available. On the other hand, all parameters of non-blocking operators are partitionable. For example, all source operators are non-blocking. Operators that are neither blocking nor non-blocking are called partially blocking. For example, the difference operator computes the difference of two sets of tuples S_1 and S_2 of the same kind. The first set of tuples S_1 is partitionable whereas the second set of tuples S_2 is not. The operator needs to know all sets of tuples S_2 that must appear in S_1 before returning any (partial) results. The partially blocking is also considered as blocking operator in this paper. In DeCloud, there are other six operators besides two dummy operators: start and end, they are filter, project, join, difference, union, rename and dispatch where join operator, difference operator and dispatch operator are the blocking operators. In addition, dummy operator is considered as blocking operator.

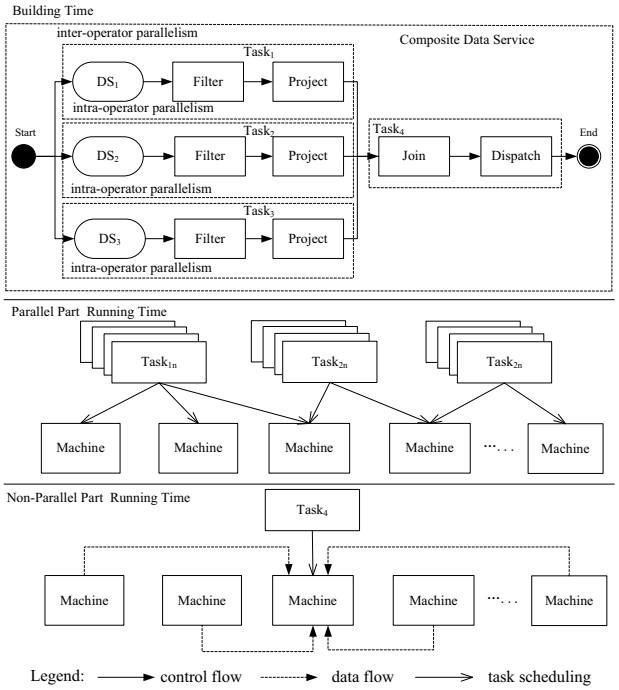


Figure 2. Transformation of Composite Data Service into Independently Executable Tasks

The advantages of partitionable input data for an operator are twofold. First, an operator can partition each partitionable parameter and thereby split the operator execution into multiple tasks that can be executed in parallel (intra-operator parallelism). The complete operator result can later be reconstructed from all task results. On the other hand, data partitioning can also be used for asynchronous (pipelined) operator execution. Individual task results, i.e., partial operator results, can already be handed off (or "pushed") to its succeeding operator (which in turn may already produce a partial result) if the corresponding input parameter of the succeeding operator is partitionable, too. This method therefore allows for an overlapping execution of neighboring operators in the dataflow graph. Finally, independent operators, i.e., operators that neither directly nor indirectly relies on each others' outputs, are run in parallel (inter-operator parallelism).

The execution thus entails the correct transformation of a composite data service into a series of independently executable tasks, as shown in Figure 2. The upper part shows the dataflow of a composite data service, where the DS_1 provides the latest movie information, such as movie name, theater location, and show time, the DS_2 provides the coordinates of each location, and the DS_3 provides the weather information. The three data services are transformed and joined to present an integrated view of the latest movie directory.

In the composite data service, there are four blocking operators: start, join, dispatch, end, they partition the dataflow into two parts, the first part is called parallel part, which is from start to join, and the second part is called non-parallel part, which is the combination of join, dispatch, and end. The middle part and bottom part of the Figure 2 show transformed tasks of the two parts. In running time, each part is executed in sequence, and their branches are applied on the following rules:

- In parallel part, if there is only one branch, then the branch is transformed into a series of parallel executable tasks.
- In parallel part, if there is more than one branch, then these branches are transformed into a series of independently executable tasks, where each branch is transformed into a series of parallel executable tasks. For example, DS_1, DS_2, and DS_3 are transformed into three independently executable tasks $Task_1$, $Task_2$, $Task_3$ respectively, and $Task_1$ can be transformed into a series of parallel executable tasks named $Task_{11}$, $Task_{12}$,…, $Task_{1n}$.
- In non-parallel part, there is only one task, and it is executed after all input parameters are generated. For example, $Task_4$ is executed after all input parameters are generated.

In following section, two scheduling algorithms for tasks generated from parallel part and non-parallel part are proposed to minimize the usage cost required to complete the execution of composite data service when computational capability is charged.

III. PROBLEM DESCRIPTION

There are two basic scenarios involved in the Figure 2, and we try to solve problems related with those scenarios. Such problems are all about cost optimization over multiple machines when computational capability is charged.

Scenario A: Minimize the usage cost when tasks generated from parallel part are executed. In this scenario, we suppose that there is large number of machines available with different usage prices provided by cloud services, each machine has a computational capacity, the task can smoothly be executed only the resource requirement is satisfied. Generally, the more the computational capability of the machine is, the higher the usage price of the cloud service is. Each task t_i requires C_i-weighed computational resources so that it can be executed correctly. When the tasks are execution, our problem is how they are scheduled intelligently to appropriate machines to make the minimum cost.

Since cloud services provide massive computational resources, the total computational capacity exceeds the total requirement by far, so there are enough machines available when a group of tasks comes. Suppose we spend 5$ to allocate some machines to complete these tasks; But if 2$ is ok, then we choose 2$ and thereby 3$ is left.

EXAMPLE 1. Consider the scheduling plan in Figure 2. Let the requirement of the $Task_1$, $Task_2$, $Task_3$ and the capability of machines to be as follows:

i	1	2	3
Requirement of $T_i(C_i)$	6	2	3
j	1	2	3
Available Capability (L_j)	8	5	4
Computing Price (P_j)	1.5	1	1.2

Then if we dispatch each task to a different machine (it is possible because L_j is greater than C_i), the above task scheduling process will be generalized into the multiple-choice knapsack problem, and the problem description is as follows: Suppose there are *n* tasks and *m* machines, the scheduling plan is T, where T_{ij}=1 indicates the i_{th} task is scheduled to the j_{th} machine and T_{ij}=0 indicates the i_{th} task is not scheduled to the j_{th} machine, the 0-1 programming model is as shown in expression 1. Comparing to classical multiple-choice knapsack model, our model requires all tasks must be scheduled to machines.

$$\begin{aligned} &\min \ \sum_{i=1}^{n}\sum_{j=1}^{m} C_i \times T_{ij} \times P_j \\ &st. \quad \sum_{i=1}^{n} C_i \times T_{ij} \le L_j, 1 \le j \le m \\ &\qquad \sum_{j=1}^{m} T_{ij} = 1, T_{ij} \in \{0,1\}, 1 \le i \le n \end{aligned} \tag{1}$$

Scenario B: Minimize the usage cost when tasks generated from non-parallel part are executed. The scenario is independent with scenario A. When the task is scheduled to machine and ready for execution, all input parameters must be guaranteed to be available, so the usage cost not only contains the computation cost, but also contains the transfer cost, and we have to explore the best tradeoff between them.

EXAMPLE 2. Consider the scheduling plan in Figure 2. Suppose the tasks $Task_1$, $Task_2$, $Task_3$ output three 1MByte

parameters after execution, and are placed the 1_{th} and the 2_{th} machine. The requirement of $Task_4$ and the computational capacity and the usage price of each machine are showed as follows:

I	4		
Requirement of T(C)	2		
J	1	2	3
Available Capability (L_j)	2	0	4
Computing Price (P_j)	1.5	1	1.2
Transfer Price (CP_{ij})	1		

And now we have two plans for this problem: (a) allocate the 1_{th} machine to run $Task_4$; (b) allocate the 3_{th} machine to run $Task_4$. In this scenario, we can easily tell that plan (b) is better, this is because the sum of the transfer cost (1+1)/1=2 and the computation cost $1.2\times 2=2.4$ of plan (b) is less than the sum of transfer cost (1+1)/1=2 and the computation cost $1.5\times 2=3$ of plan (a).

Suppose the scheduling plan is T, where $T=(T_1,T_2,\ldots,T_m)$ and $T_j=1$ indicates the *task* is scheduled to the j_{th} machine and $T_j=0$ indicates the *task* is not scheduled to the j_{th} machine. T_j' is the negate of T_j. The CP indicates the transfer price and the value is a constant to simplify the computation. Parameter is the input parameters for *task*, and they are distributed different machines. Parameter[j] indicates the size of input parameters located in the j_{th} machine, besides, other symbols are the same as scenario A. The 0-1 programming model is as shown in expression 2.

$$\begin{aligned} &\min \ \sum_{j=1}^{m} C\times T_j\times P_j+\sum_{j=1}^{m}(T_j'\times Parameter[j])/CP \\ &st. \quad C\times T_j\le L_j, 1\le j\le m \\ &\qquad \sum_{j=1}^{m} T_j=1, T_j\in\{0,1\}, 1\le i\le n \end{aligned} \tag{2}$$

IV. ALGORITHMS FOR EXECUTION PLANS

A. Optimal Execution Plans for Scenario A

The problem in Scenario A is NP-hard problem, and the computational complexity is 2^{mn}. Generally, the approximate algorithms are suitable for the problem. In this paper, we use genetic algorithm[6], one of the approximate algorithms, to solve the problem. The process is as follows:

- Generic Encoding: The generic algorithm for multiple-choice knapsack problem usually uses the binary encoding as generic encoding. Considering the characteristic that the solution space can be mapped into the matrix, we use the matrix $T(m\times n)$ as the genetic encoding of the problem, where $T_{ij}=1$ indicates the i_{th} task is scheduled to the j_{th} machine and $T_{ij}=0$ indicates the i_{th} task is not scheduled to the j_{th} machine.
- Fitness Function: According to expression 1, the fitness function is as follows:

$$f(T)=\sum_{j=1}^{m} J_j(T)+\sum_{j=1}^{m} G_j(T)+\sum_{i=1}^{n} H_i(T)$$

Where the object function of the j_{th} machine is as follows:

$$J_j(T)=\sum_{i=1}^{n} T_{ij}\times P_j\times C_i$$

And the punish function that exceeding the computational capacity of the j_{th} machine is as follows, where α_j is the punishing coefficient:

$$G_j(T)=\begin{cases}\alpha_j(L_j-\sum_{i=1}^{n} T_{ij}\times C_i), & L_j\le\sum_{i=1}^{n} T_{ij}\times C_i\\ 0, & L_j\ge\sum_{i=1}^{n} T_{ij}\times C_i\end{cases}$$

And the punish function that one task is scheduled more than one or no machine is as follows, where β_i is the punishing coefficient:

$$H_i(T)=\begin{cases}\beta_i(1-\sum_{j=1}^{m} T_{ij}), & \sum_{j=1}^{m} T_{ij}\ge 1\\ 0, & 0<\sum_{j=1}^{m} T_{ij}\le 1\\ \beta_i, & \sum_{j=1}^{m} T_{ij}=0\end{cases}$$

It should be noted that if $J(T)$ is negative in some cases, then $J(T)$ is changed to 0. As we know, the generic algorithm mainly includes the selection, crossover, mutation. Since the length of encoded string is $m\times n$, if we use the single-point crossover, the upper bound of total variance distance between the probability distribution vector of the population and the corresponding steady-state distribution vector[7] will become greater, so that the speed of convergence to steady state will be slower.

To solve this problem, we use the shield character. The shield character is randomly generated and has the some structure as the individual' gene. The process of crossover is as follows: Two parent individuals T_1 and T_2, if shield character corresponding to the ij gene $M_{ij}=0$, then there is no crossover and $T_{1ij}'=T_{1ij}$, $T_{2ij}'=T_{2ij}$, on the contrary, if shield character $M_{ij}=1$, the corresponding two genes cross each other and $T_{1ij}'=T_{2ij}$, $T_{2ij}'=T_{1ij}$.

The algorithm is shown and can get an approximate optimal plan in O(Generation × Num × m × n) time, so is polynomial algorithm.

Generic Searching Algorithm(GSA)

Input: $m\leftarrow number\ of\ execution\ engines$, $n\leftarrow number\ of\ tasks$
$g\leftarrow 0, Generation, Num, bestfit\leftarrow 0,$
$P_c\leftarrow crossover\ probability, P_m\leftarrow mutation\ probability$

Output: $\cos t$

1. $POP(g)=Initialize(mn, Num)$
2. $for(i=0.i\le Num, i++)$
3. $f_i=J(POP_i(g))$
4. $\cos t\leftarrow \min(f), g++$
5. $while(g<Generation)$
6. $for(j=0, j\le Num, j++)$
7. $P_i=f_i/\sum_{i=0}^{N} f_i$
8. $POP(g)=NewPOP(P)$
9. $POP(g)=CrossPOP(POP(g), P_c)$
10. $POP(g)=MutPOP(POP(g), P_m)$
11. $for(i=0.i\le Num, i++)$
12. $f_i=J(POP_i(g))$
13. $\cos t\leftarrow \min(f), g++$
14. $return\ \cos t$

B. *Optimal Execution Plans for Scenario B.*

Conditions involved in scenario B are different from that of scenario A, and the problem is simplified, so we can use heuristic searching algorithm to solve the problem. In this algorithm, the heuristic rules can reduce the unnecessary computation. The heuristic searching algorithm computes an optimal plan in $O(m^2)$ time and is polynomial algorithm.

Heurist Searching Algorithm(HSA)

Input: $m \leftarrow$ *number of* available *execution engines*, *task*

Output: $\cos t$

1. *for* $(i = 1, i \le m, i++)$
2. *if* $(task.C \le L_i)$
3. $\cos t_1 = task.C \times P_i$
4. *if* $(\cos t_1 > \cos t)$ *continue*;
5. *for* $(j = 1, j \le m, j++)$
6. $\text{cost}_1 += Parameter[j] / CP_{ij}$
7. *if* $(\cos t_1 > \cos t)$ *break*;
8. $\cos t = \cos t_1$
9. *return* $\cos t$

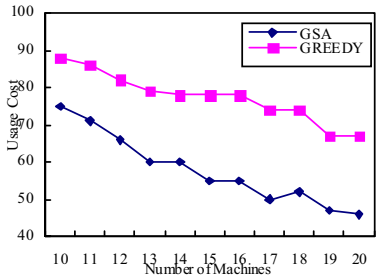
V. IMPLEMENTATION AND EXPERIMENTS

We implemented an initial prototype, described in Section 2. Here we report on a few experiments with it. Not surprisingly, in our experiments, execution plan of composite data service reflects our theoretical results (thereby validating the cost saving). Using usage cost as metrics, we compared the execution plan produced by our optimization algorithm against the plans produced by the following simpler algorithms.

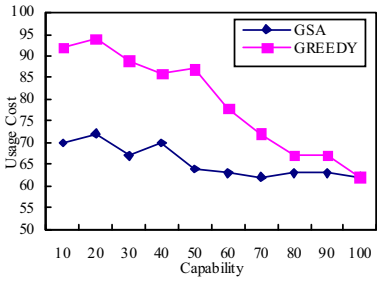
- Greedy: This algorithm attempts to exploit the minimum cost by dispatching tasks to machines with the minimum cost whenever possible. For example, if a task T_i is supposed to require 40 resources, and the available computational capability of the k_{th} machine L_k is 46 and the P_k is the minimum, then T_i would be dispatched to the machine. Greedy is used to finish a comparison with our algorithm in the first scenario that mentioned before.
- Random: The tasks are scheduled randomly. Random is used to finish a comparison with our algorithm in the second scenario that mentioned.

The experimental setup consists of two parts: the client side, consisting of spreadsheet client interface, and the server side, consisting of virtual machines provided by Eucalyptus[1]-an open source cloud platform set up by us. Our setup prototype is a multithreaded system written in Java. It implements the two core scheduling algorithms we proposed in this paper. For communicating with data services using SOAP, our prototype uses Codehaus XFire tools. The function of executing a data service is realized inside execution engines in virtual machines. We can create and delete a virtual machine that has specific properties on demand: computational capability and computing price. We use Jetty as the application server and Kettle[8] tools for data service composition. Each of our experimental data services runs on different machines. We develop some data services, and compose a composite data service just as shown in Figure 2. The composite data service is transformed into 10 tasks, and their requirement ranges from 1 to 10. When scheduling these tasks, we dynamically set the computing price of machines and their computational capability, and collect the transfer price through the Profiling and Statistics component.

When the number of machine increases from 10 to 20, surprisingly, we find that the number of machines required is nearly the same when using GSA and GREEDY, while the usage cost is completely different as shown in Figure 3(a). Figure 3(a) shows that, the advantage of GSA mounts up as the number of machine increases. In the extreme case, when the computation capacity is increased to exceed the total requirements of all tasks, the plans generated from GSA and GREEDY are the same, so we can conclude that the more the computation capacity, the closer the usage cost. The following experiment is used to validate the conclusion. In Figure 3(b), the number of machines is 10, we find only when the computational capability is higher do the GREEDY obtain the similar cost as GSA. The experimental results are the same with the conclusion.



(a)



(b)

Figure 3. Comparison of Usage Cost in Scenario A

See the results produced in Figure 4. Not surprisingly, the usage cost descends as the number of machines increase, no matter which algorithms were applied.

Nevertheless, results obtained from HSA were always more excellent than that from RANDOM. The main reason is the HSA reduces the unnecessary transfer cost.

1 http://www.eucalyptus.com/

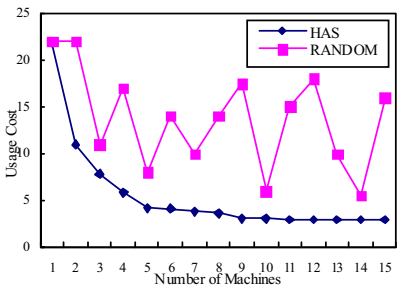


Figure 4. Comparison of Usage cost in Scenario B

VI. RELATED WORKS

In client end, mashup-based data integration has become very popular in recent years and many tools and frameworks have been developed. We[9] divided these tools into four categories: 1) flow-chart-like programming tools, such as Yahoo Pipes; 2) spreadsheet-like programming tools, typical representatives are SpreadMash[10]; 3) tree-based programming tools, MashMaker[11] falls into this category; 4) browser-centric programming tools, d.mix[12] belongs to this category. Lorenzo[13] has analyzed the advantages and disadvantages of these tools, but there are seldom works focusing on the performance. A few recent mashup platforms also deal with mashup efficiency. CoMaP[14] targets a distributed mashup execution that minimizes the overall mashup execution time of multiple hosted mashups. The AMMORE[15] system even modifies original mashup dataflows to avoid duplicate computations and unnecessary data retrievals.

In server end, data integration is experienced parallel data processing in a wide variety of applications. Driving forces are simple and powerful parallel programming models such as MapReduce[16] and Dryad[17]. Although the MapReduce program model is limited to two dataflow primitives (map and reduce), it has proven to be very powerful for a wide range of applications. On the other hand Dryad supports general dataflow graphs. Freely available frameworks such as Hadoop further stimulate the popularity of distributed data processing. Higher-level languages can be layered on top of these infrastructures. Examples include the high-level dataflow language Pig Latin[18], DryadLINQ[19] as well as SCOPE[20] that offers a SQL-like scripting language on top of Microsoft's distributed computing platform Cosmos. Similar to DeCloud, a high-level program is decomposed into small buildings blocks (tasks) that are transparently executed in a distributed environment. But there is no consideration how to reduce the usage cost when cloud computing provides good enough performance.

VII. CONCLUSIONS AND FUTURE WORKS

We presented a cloud-based data integration system-DeCloud, DeCloud' task-based execution approach allows for an efficient and parallel execution and is tailored to recent cloud-based web instruments. We have devised new algorithms to minimize the cost required to complete the composite data service when computational capability is charged. Our algorithms currently just simply expect that the transfer cost is equivalent the same. However, as the case stands, the transfer cost might obviously vary. We will consider the case in future works.

ACKNOWLEDGMENT

The research work is supported by the National Natural Science Foundation of China under Grant No. 60970131, No.61033006, and No.60903048.

REFERENCES

[1] Li X H, Li Y, Liu T C, Qiu J, Wang F C. The Method and Tool of Cost Analysis for Cloud Computing. CLOUD 09, 2009, 93-100.

[2] Weiss A. Computing in the Cloud. ACM Networker, 2007,11:18-25.

[3] Li W B, Zhao Z F, Fang J, Chen K. Cost optimization for Composite Services Through Multiple Engines. ICSOC, 2007:595-605.

[4] Hoyer V, Fischer M. Market overview of enterprise mashup tools. ICSOC, 2008:708-721.

[5] Zhang P, Wang G L, Ji G, Liu C. Optimization Update for Data Composition View Based on Data Service. Chinese Journal of Computers, 2011.34(12):2344-2354.

[6] Andreas T, Erhard R. CloudFuice: A Flexible Cloud-Based Data Integration System. ICWE, 2011:304-318.

[7] John, H. Adaptation in natural and artificial systems.1992.MIT Press, Cambridge, MA.

[8] Suzuki J . A Markov chain analysis on simple genetic algorithms. IEEE Trans Systems Man Cybernet,1995(25):655-659.

[9] Casters M, Bouman R,Dongen J. Pentaho Kettle solutions: building open source ETL solutions with Pentaho Data Integration. 2010, Wiley Publishing.

[10] Wang G, Yang S, Han Y. Mashroom: End-User Mashup Programming Using Nested Tables. WWW, 2009, 861-870.

[11] Kongdenfha W, Benatallah B, Saint-Paul R, et al. SpreadMash: A Spreadsheet-Based Interactive Browsing and Analysis Tool for Data Services. CAiSE, 2008, 343-358.

[12] Ennals R J, Garofalakis M N. MashMaker: Mashups for the Masses. SIGMOD, 2007, 1116-1118.

[13] Hartmann B, Wu L, Collins K, et al. Programming by a Sample: Rapidly Creating Web Applications with d.mix. UIST, 2007, 241-250.

[14] Di Lorenzo, G., Hacid, H., Paik, H., Benatallah, B. Data integration in mashups. SIGMOD, 2009, 38(1), 59-66.

[15] Hassan O, Ramaswamy L, Miller J. Comap: A cooperative overlay-based mashup platform. CoopIS, 2010, 454-471.

[16] Hassan, Ramaswamy, Miller. Enhancing Scalability and Performance of Mashups Through Merging and Operator Reordering. ICWS, 2010, 171-178.

[17] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. Communications of the ACM, 2008, 51(1):107-113.

[18] Isard M, Budiu M, Yu Y, Birrell A, Fetterly D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. ACM SIGOPS Operating Systems Review, 2007, 41(3):59-72

[19] Olston C, Reed B, Srivastava U et al. Pig Latin: A Not-So-Foreign Language for Data Processing. SIGMOD, 2008, 1099-1110.

[20] Yu Y, Isard M, Fetterly D et al. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. OSDI, 2008, 1-14.

[21] Chaiken R, Jenkins B, Larson P et al. SCOPE: Easy and efficient parallel processing of massive data sets. VLDB, 2008, 1(2):1265-1276