An Efficient Data Maintenance Strategy for Data Service Mashup Based on Materialized View Selection

Peng Zhang[1,2], Yanbo Han[2], and Guiling Wang[2]

[1] North China University of Technology, 100041, Beijing, China
[2] Institute of Information Engineering, Chinese Academy of Sciences, 100093, Beijing, China
{zhangpeng,wangguiling}@software.ict.ac.cn, yhan@ict.ac.cn

Abstract. While end-users enjoy the full-fledged data service mashup with high convenience and flexibility, such issues as the response efficiency and the maintenance cost have popped up as the major concerns. In this paper, an efficient data maintenance strategy for the data service mashup is proposed. The strategy proposes a data maintenance model to measure the response cost and update cost of a group of data service mashups in terms of the request frequency and update frequency. Based on the model, a materialized view selection for data service mashup is proposed. Experiments show that our strategy can effectively reduce the maintenance cost of a lot of hosted data service mashups.

Keywords: Data Mashup, Data Service, Data Maintenance, Materialized View Selection.

1 Introduction

Data service mashup has become so popular over the last few years. It is a special class of mashup application that combines information on the fly from multiple data sources. Its applications vary from addressing transient business needs in modern enterprises to conducting scientific research in e-science communities [1]. The data sources are provided through Web Services, also known as DaaS (Data-as-a-Service) or Data Service [2, 3]. While providing enhanced immediacy and personalization to explore, aggregate and enrich data from various heterogeneous sources, data service mashups also pose distinct data maintenance challenges. In general, the mashup results are often cached in order to enhance the performance of responding to end-users' request. However, when the primitive data sources are updated, the cache should be updated to ensure their consistency with the underlying data sources, which brings the high maintenance cost. For a mashup platform which hosted a lot of mashup applications, some cached results may be shared and reused by other mashups. So it will be more efficient to materialize certain "shared" mashup results to achieve the best performance with minimum consistency maintenance cost. In this paper, we call this problem as the data maintenance problem. The challenges to solve this problem are analyzed from the following three aspects:

A. Ghose et al. (Eds.): ICSOC 2012, LNCS 7759, pp. 14–23, 2013.

Firstly, most of the mashup platforms allow all kinds of users to develop their own mashups, so there are often a large number of data service mashups hosted on the mashup platforms, which implies that the data maintenance cost for mashup platforms is much higher.

Secondly, data service mashups are often designed by non-technical-savvy end-users, and hence they are not necessarily optimized from the point of view of data maintenance.

Thirdly, the data sources often have various non-functional characteristics such as update frequency, data volume and so on but the data service mashups lack of their description. So they can not optimize the data maintenance utilizing these characteristics.

All these reasons make it quite necessary to consider the data maintenance of a lot of hosted data service mashups. Unfortunately, the efficiency aspects of mashup platforms have not received enough attention from the research community. Only a few works give some preliminary research results [4, 5]. In this paper, we present an efficient data maintenance strategy for data service mashup, which has the following special features.

- We establish the maintenance cost model for the data service mashup, and propose the materialized view selection strategy based on the maintenance cost model.
- We evaluate the performance of our proposed strategy through standard datasets from TPC-H and demonstrate that our strategy achieves significant improvements.

The paper is organized as follows: Section 2 gives an example to formulate the problem. Section 3 introduces the maintenance cost model and materialized view selection strategy. A detailed example is introduced. Section 4 is experiment and discussion. Section 5 introduces related works. Section 6 sums up with several concluding remarks.

2 Problem Description

Consider an example data service mashup. Firstly, it invokes data service M_1 and M_2, fetches the popular movies from mtime.com and movie.hao123.com (two famous movie guide portal in China). Secondly, the output of M_1 and M_2 are combined as $M_1 \cup M_2$. Thirdly, it invokes the movie review API from douban.com (the most popular book & movie reviews portal in China). The result is a complete popular movie list with reviews as $(M_1 \cup M_2) \triangleright_{Ri} R$. The mashup steps are shown in Figure 1, where the data services from the two popular movie websites are abbreviated as M_1 and M_2, and the data service from the movie review website is abbreviated as R.

In fact, in order to get the same results, different users may use different mashup scheme. The right side of Figure 1 shows the other mashup scheme. In this scheme, at first, the movie lists from mtime.com and movie.hao123.com are each used as the input to invoke the API from douban.com. Then both the outputs are combined. The outputs of both mashups are the same.

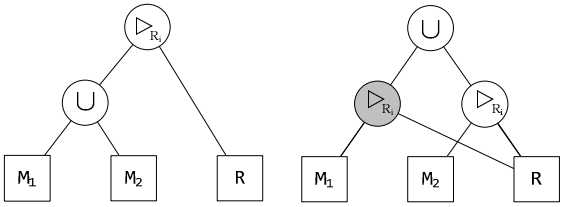


Fig. 1. An example data service mashup

According to the interaction between user and the data mashup, we can make a distinction of the execution mode of mashup between active and passive as shown in Table 1. At the active mode, when some data sources are updated, the platform actively sends the update to users. At this mode, we design a "materialized data view" to cache the results on the client side. As such, the maintenance cost of a mashup in active mode is equal to the cost of responding to the update of the data sources.

While for those mashups in passive mode, they send the results to users only when there comes in a user request. At this mode, there is no materialized data view for the mashup. The mashup gets results by re-computing the underlying data sources from bottom to top. Compared with the "materialized data view" to cache the results, it can be called the "virtual data view". In order to make it more efficient, the mashup can reuse some materialized results. As such the maintenance cost of a mashup in passive mode is equal to the cost of responding to request.

Table 1. The maintenance cost category

Execution Mode	View Mode	Maintenance Cost
Active	Materialized	update cost of the data sources
Passive	Virtual	response cost of the user's request

Therefore, given a group of mashups, the data maintenance cost is composed of two parts: one part is the response cost of all virtual data views, and the other part is the updating cost of all materialized data views, where some virtual data views can reuse some materialized execution results of the materialized data views.

When the outputs of several mashups are the same, their execution costs may not be the same from the following analysis. For example, as shown in the right part of Figure 1, if $M_1 \triangleright_{Ri} R$ is executed in active mode, and $(M_1 \triangleright_{Ri} R) \cup (M_2 \triangleright_{Ri} R)$ is executed in passive mode, then $(M_1 \triangleright_{Ri} R) \cup (M_2 \triangleright_{Ri} R)$ can reuse the materialized execution result of $M_1 \triangleright_{Ri} R$ to reduce response cost, so that the data maintenance cost is reduced. However, because there are no common part between the mashup shown in the left part of Figure 1 $(M_1 \cup M_2) \triangleright_{Ri} R$ and $M_1 \triangleright_{Ri} R$, so the $(M_1 \cup M_2) \triangleright_{Ri} R$ can not reuse the materialized execution result of the mashup $M_1 \triangleright_{Ri} R$, and maybe generate higher data maintenance cost. So in Figure 1, though $(M_1 \triangleright_{Ri} R) \cup (M_2 \triangleright_{Ri} R)$ has the same results as the $(M_1 \cup M_2) \triangleright_{Ri} R$, the data maintenance costs are different.

In addition, for single mashup, the mashup schema also has an impact on the data maintenance cost. For instance, one schema first filters the popular movie list from "mtime.com" according to the category, and then sorts the list by their score. The other schema sorts the list ahead of filtering. The former one's data maintenance cost is lower than the latter one even if they have the same results. This is because when data sources are updated, filtering generally makes the intermediate results for computation reduced.

Based on above analysis, we can get such a conclusion that given that different mashup schemas and execution modes, even if the mashups output the same results, their data maintenance cost may be different, so our goal can be defined as follows: Given a group of mashups, how to select the mashup schema and execution mode, in order to get the minimum data maintenance cost.

3 Materialized View Selection

The mashup operators include *count*, *filter*, *union*, *unique*, *join*, *sort*, *projection*, *truncate*, *tail*, *rename*, which are similar to the operators in the relational algebra, and we use Σ, σ, $\cup$, μ, $\triangleright_j$, s, π, t, τ, r respectively to represent them, so the mashup can represented by tree that could be transformed into equivalent results according to the equivalent tranformation rules [6]. In order to select mashup schema and execution mode for each mashup to get the minimum data maintenance cost, our solution includes two steps: the first step is to establish the maintenance cost model for the mashups; the second step is to solve the model to get the optimization results. Here, we still take the two mashups $(M_1 \cup M_2) \triangleright_{Ri} R$ and $M_1 \triangleright_{Ri} R$ for example to firstly introduce their data maintenance cost model.

3.1 Maintenance Cost Model

For simplicity, we neglect the join attribute in this example, and use the upper case letter to represent the name of set or function, and use the lower case letter to represent the element of the set, where the length of set indicates the number of elements. For example, $V=\{(M_1 \cup M_2) \triangleright_{Ri} R, M_1 \triangleright_{Ri} R\}=\{v_1,v_2\}$, represents a set of mashups, and their maintenance cost model as shown in formula 1:

$$c_V=\min_{P \cup A=V' \wedge V' \in E(V)} (c_P+c_A) \quad (1)$$

Where P represents the set of mashups with the passtive mode, A represents the set of mashups with the active mode, c_V represents the data maintenance cost of the mashups, c_P represents the response cost of the mashups with the passive mode and will be introduced in following section, c_A represents the update cost of the mashups with the active mode and will be introduced in following section, and V' is an equivalent set of V. In order to calculate them, we first have to obtain the following three matrices EV, BE, AB. At the same time, some symbols, such as logic negative "¬", logic and "∧", dot product "⊗", matrix multiplication "×", matrix or vector transpose "T", and the function L changing any non-zero to zero are introduced.

$E(v_i)$ is the set of the mashups which are equivalent to the mashup v_i. $E(v_i)$ can be obtained by equivalence transformation rules. E represents all mashups which are equivalent to any element in the set V, and we call any element of E equivalent mashup. In this example, $E(v_1)=\{(M_1 \cup M_2) \rhd_{Ri} R, (M_2 \cup M_1) \rhd_{Ri} R, (M_1 \rhd_{Ri} R) \cup (M_2 \rhd_{Ri} R), (M_2 \rhd_{Ri} R) \cup (M_1 \rhd_{Ri} R)\}=\{e_1,e_2,e_3,e_4\}, E(v_2)=\{M_1 \rhd_{Ri} R\}=\{e_5\}, E=\{e_1, e_2, e_3, e_4, e_5\}$. EV is a 0-1 Boolean matrix with $|V|\times|E|$, and if and only if $E(v_i)=e_j$, $EV(i, j)=1$, else $EV(i, j)=0$. In this example,

$$EV = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$B(e_i)$ is the set of all intermediate results of the equivalent mashup e_i. B represents the set of all intermediate results of all equivalent mashups in E, and we call any element of B intermediate result. In this paper, we only consider the re-computing cost of the intermediate results. In fact, there is also communication cost, but it has no impact on our model. In this example, $B=\{M_1 \cup M_2, M_2 \cup M_1, M_1 \rhd_{Ri} R, M_2 \rhd_{Ri} R, (M_1 \cup M_2) \rhd_{Ri} R, (M_2 \cup M_1) \rhd_{Ri} R, (M_1 \rhd_{Ri} R) \cup (M_2 \rhd_{Ri} R), (M_2 \rhd_{Ri} R) \cup (M_1 \rhd_{Ri} R)\}$. BE is a 0-1 Boolean matrix with $|E|\times|B|$, if and only if $B(e_j)=b_i$, $BE(i, j)=1$, otherwise $BE(i, j)=0$. In this example,

$$BE = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

$A(b_i)$ is the set of all atomic data services related with the intermediate result b_i, and we call any element of A atomic data service. In this example, $A=\{M_1,M_2,R\}$. AB is a 0-1 Boolean matrix with $|B|\times|A|$, and if and only if $A(b_j)=a_i$, $AB(i, j)=1$, otherwise $AB(i, j)=0$. In this example,

$$AB = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}^T$$

p^V is a 0-1 Boolean vector representing the execution modes of the mashups, where 1 represents the mashup is executed in active mode, and 0 represents the mashup is executed in passive mode. r^V and u^A are the request frequency vector of the mashups and the update frequency vector of atomic data services respectively. They can be obtained from monitoring and service interface description. In order to facilitate the computation of the data maintenance cost, in this example, we assume that $r^V=[1,1]$, and $u^A=[1,1,1]$. c^B is the cost vector of the single-step-computing of intermediate results. For example, the cost of the single-step-computing of the $(M_1 \cup M_2) \rhd_{Ri} R$ is the cost of join operation between $M_1 \cup M_2$ and R. The c^B can be obtained by testing. In this example, we suppose

that c^B=[1,1,2,2,3,3,1,1]. The cost of the single-step-computing of $(M_1 \cup M_2) \triangleright_{Ri} R$ is set to 3, which is lower than the sum of $M_1 \triangleright_{Ri} R$ (set to 2) and $M_2 \triangleright_{Ri} R$ (set to 2), and higher than any of them. The above information is all input to establish the data maintenance cost model of the mashups. Next, we will explain how to compute the c_P and c_A in our example.

$c_P=((((r^V \times EV) \otimes x^E) \times BE) \otimes c^B) \times (L(x^E \times BE) \wedge \neg x^{BP})^T$ is the total cost of the single-step-computing of the intermediate results, where x^E is a 0-1 vector, 0 indicates that the equivalent mashup is not selected and 1 indicates the equivalent mashup is selected. The $(r^V \times EV) \otimes x^E$ is the request frequency vector of the selected equivalent mashups, and the $((r^V \times EV) \otimes x^E) \times BE$ indicates the number of the single-step-computing of the intermediate results of the selected equivalent mashups. After the $((r^V \times EV) \otimes x^E) \times BE$ is multiplied by c^B, we can get the total cost of the single-step-computing of the intermediate results without taking into account reusing the materialized results. The total cost is multiplied by the transpose of $L(x^E \times BE) \wedge \neg x^{BP}$ to eliminate the cost of the single-step-computing of materialized results, where the $x^{BP}=L((p^V \wedge x^E) \times BE)$ represents the materialized results of all selected equivalent mashups.

$c_A=((x^{BP} \otimes c^B) \times AB) \times (u^A)^T$ is the total cost of the single-step-computing of the intermediate results related with the atomic data services when the atomic data services are updated. Firstly, we multiply c^B and x^{BP} to get the total cost of all single-step-computing of intermediate results. AB represents the relationships between atomic data services and intermediate results, which is multiplied by $x^{BP} \otimes c^B$ to get the total cost of the all single-step-computing of intermediate results when all atomic data services are updated. The total cost further is multiplied by the transpose of u^A to get the final total update cost, where u^A is the update frequency vector of all atomic data services. In this example, we use a genetic algorithm as shown in [3] to solve the 0-1 programming to get the approximate optimal solution, and the solution is as follows: p^V=[0,1] and x^E=[0,0,1,0,1], namely the $(M_1 \cup M_2) \triangleright_{Ri} R$ and $M_1 \triangleright_{Ri} R$ is passive mode and active mode respectively, and the script of $(M_1 \cup M_2) \triangleright_{Ri} R$ is replaced with the script of $(M_1 \triangleright_{Ri} R) \cup (M_2 \triangleright_{Ri} R)$.

4 Experiment and Discussion

Our experimental setup is based on our data service mashup platform-DSM [7]. We simulate a group of data services spread out on the Internet based on TPC-H[1]. TPC-H describes a multi-part production and sales scenario, involving eight data tables. The tables are encapsulated into a group of Internet accessible data services. Since the experiment only considers the number of mashups, the request frequency of mashups, and the update frequency of the atomic data services, the data size of the output of the atomic data services is kept unchanged. We configure the default value to be 100 KB. In the future, we will consider the dynamic data size of data services.

[1] http://www.tpc.org/tpch/

In the following experiments, we build 10 mashups. To simulate the real data service mashups, the average number of operators in one data service mashup is limited less than 8, which is similar to those mashups on Yahoo! Pipes [8]. To simulate the real large-scale mashup platform scenario, we duplicate 80 mashups according to the Zipfian distribution with α=0.9 based on the statistical observation from syndic8, which has been studied in paper [4].

We use JMeter[2] to simulate the stress testing on a single machine with 2GB memory and 2.26GHz CPU. In following experiments, we quantify the performance benefits of the materialized selection strategy. The materialized selection strategy is compared with two other strategies: "All Materialized" and "No Materialized", the former means all results are materialized and the latter means no result is materialized. These three strategies are compared with respect to the total cost incurred by the mashup platform in serving the end-user requests. For an individual mashup, the cost is quantified as the associated computational latency at the mashup platform. In Figure 2, we randomly select mashups, and compare the data maintenance cost of three strategies as the number of the mashups increases from 4 to 80. The mean of the request frequencies is set to 50, and the mean of the update frequencies is set to 5. As the results indicate, the data maintenance cost incurred by DSM is lower than the other two strategies throughout the simulated range. The most important thing is DSM's curve shows log tendency rather than linear tendency, and the reason is that the probability of reuse increases as the number of the mashups increases.

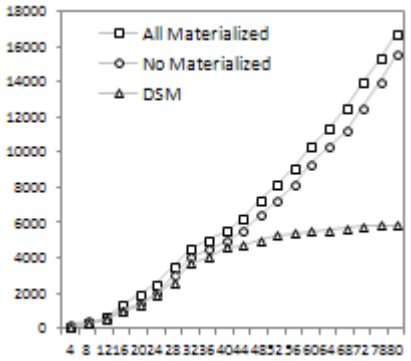
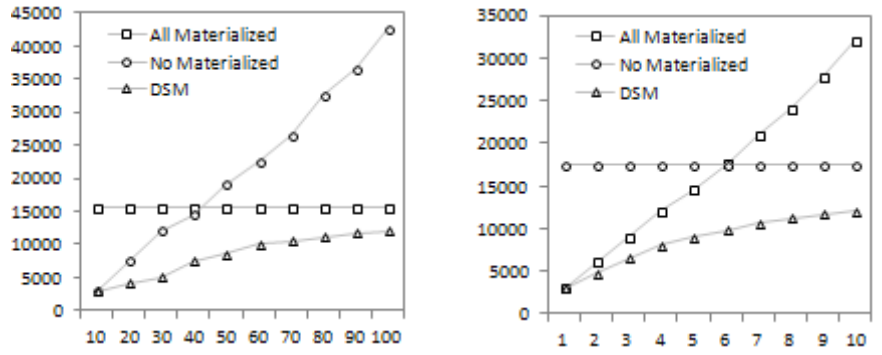


Fig. 2. The maintenance cost comparison

In Figure 3(a), we compare the three strategies as the mean of the request frequencies of all the one thousand mashups varies from 10 requests per unit time to 100 requests per unit time. The mean of the update frequencies of the data services is set to 5. The system is assumed to have enough storage to materialize the results of all mashups. Figure 3(a) shows the data maintenance cost per unit time for the results of the experiments. As the results indicate, the data maintenance cost incurred by DSM's

[2] http://jmeter.apache.org/

materialized selection is lower than the other two strategies throughout the simulated request frequency range. The data maintenance cost incurred by the "All Materialized" strategy is essentially constant as the requests are served using materialized results without additional computations. For this "All Materialized" strategy, the data maintenance cost is mainly due to re-computing of the materialized results when the associated atomic data services used in the mashups are updated. At very low request rates, the data maintenance cost of "No Materialized" strategy is comparable to those of the DSM. However, with the request frequency increases, the data maintenance cost of "No Materialized" strategy raises quickly. It is to be noted here that although the data maintenance cost of the DSM strategy increases with the increasing request frequency, its curve becomes flat once upon reaching the "All Materialized" cost levels. In Figure 3(b), we study the effect of update frequencies of data services on the data maintenance cost of the three strategies. The setup is very similar to that of the previous one except that the mean request frequency is fixed at 50 requests per unit time whereas the update frequency of all data services is varied from 1 to 10 per unit time. Again, we see that the DSM has the best performance than the other two strategies. However, in this experiment, the data maintenance cost of the "No Materialized" remains constant. This is because there are no materialized results that need to be recomputed when the data services are updated.



(a) Maintenance Cost When Request Frequency is Variable (b) Maintenance Cost When Update Frequency is Variable

Fig. 3. The maintenance cost comparison

We thus conclude that the maintenance cost reduction of the DSM system is achieved when there are higher request frequencies, update frequencies and the number of mashups.

5 Related Works

Since Harinarayan published the first paper about materialized view selection [9], the materialized view selection had been gradually warmed up in the database research, and attracted a growing number of researchers. Many research results continue to

emerge, including the static selection [10], the dynamic selection [11] and the hybrid approach [12], but all of them focus on traditional data integration on the data warehouse. On the contrary, this paper focuses on the popular data integration on the Web called data mashup. So although our maintenance cost model is similar to the MVPP framework for materialized view design [10], we consider the interaction between user and the mashup (the execution mode of mashup) which is different from MVPP.

In recent years, Web 2.0 technologies have been widely developed. There have been many tools and platforms to support just-in-time data mashup for non-professional users, however, seldom works have focused on the performance combined with maintenance. Hassan has presented a dynamic caching framework MACE [4] to improve the performance of data mashups. MACE continuously observes the execution of mashups, and collects statistics such as request frequencies, update frequencies and cost and output size values at various nodes of mashups. It then performs cost-benefit analysis of caching at different nodes of mashups, and chooses a set of nodes that are estimated to yield the best benefit-cost ratios. For each new mashup, MACE platform analyzes whether any of the cached results can be substituted for part of the mashup workflow. If so, the mashup is modified so that the cached data can be re-used. To a great extent, MACE enhances the performance of mashups, but MACE does not consider the reuse possibility through the mashup equivalence transformation. In addition, the paper [5] provides AMMORE platform to support mashups construction. AMMORE represents mashups as strings, and then it detects the longest common components across mashups, and merges mashups with other mashups by common components to minimize the total number of operators that the mashup platform has to execute. The longest common components can be used by multiple mashups. In this way, the maintenance cost of mashup platforms is reduced, but AMMORE does not establish their maintenance cost model. In fact, the DSM merges mashups based on the mainteance cost model, and also can minimize the number of maintenanced mashups.

On the mashup maintenance, the paper [13] presents techniques that help mashup developers to maintain applications by identifying when and how the original applications' UIs change, but the techniques are mainly used to search best matched widgets. Our prior work [2] introduces the maintenance cost model for the data service mashup, but the introduction is simple.

6 Conclusions and Future Works

Data service mashup is a special class of mashup application that combines information on the fly from multiple data sources. A challenging problem is how to achieve the best performance with the minimum maintenance cost.

This paper presents an efficient strategy, based on materialized view selection. The strategy use the data maintenance model to measure the response cost and update cost of a group of data service mashups in terms of the request frequency and update frequency. Based on the model, a materialized view selection for data service mashup is proposed. Experiments show that our strategy can effectively reduce the maintenance cost of a lot of hosted data service mashups.

Acknowledgements. The research work is supported by the National Natural Science Foundation of China under Grant No.61033006.

References

1. Barhamgi, M., Ghedira, C., Benslimane, D., et al.: Optimizing DaaS Web Service based Data mashup. In: SCC 2011, pp. 464–471 (2011)
2. Zhang, P., Wang, G.L., Ji, G., Han, Y.B.: An Efficient Data Maintenance Model for Data Service Mashup. In: IEEE International Conference on Services Computing (SCC 2012), pp. 699–700 (2012)
3. Zhang, P., Wang, G.L., Ji, G., Liu, C.: Optimization Update for Data Composition View Based on Data Service. Chinese Journal of Computers 34(12), 2344–2354 (2011)
4. Hassan, O.A., Ramaswarny, L., Miller, J.A.: The MACE Approach for Caching Mashups. International Journal of Web Services Research 7(4), 64–88 (2010)
5. Hassan, O.A., Ramaswamy, L., Miller, J.A.: Enhancing Scalability and Performance of Mashups Through Merging and Operator Reordering. In: Proceedings of the IEEE International Conference on Web Services, pp. 171–178 (2010)
6. Lin, H.L., Zhang, C., Zhang, P.: An Optimization Strategy for Mashups Performance Based on Relational Algebra. In: Sheng, Q.Z., Wang, G., Jensen, C.S., Xu, G. (eds.) APWeb 2012. LNCS, vol. 7235, pp. 366–375. Springer, Heidelberg (2012)
7. Han, Y., Wang, G., Ji, G., Zhang, P.: Situational data integration with data services and nested table. In: Service Oriented Computing and Application, pp. 1–22 (2012)
8. Yahoo! Pipes: Rewire the web (2011), `http://pipes.yahoo.com/`
9. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. ACM SIGMOD Record 25(2), 205–216 (1996)
10. Yang, J., Karlapalem, K., Li, Q.: Algorithm for materialized view design in data warehousing environment. In: Jarke, M., Carey, M.J., Dittrich, K.R. (eds.) Proc. of the 23rd Int'l Conf. on Very Large Data Bases (VLDB 1997), pp. 136–145. Morgan Kaufmann Publishers, Athens (1997)
11. Kotidis, Y., Roussopoulos, N.: A case for dynamic view management. ACM Trans. on Database Systems 26(4), 388–423 (2001)
12. Shah, B., Ramachandran, K., Raghavan, V., Gupta, H.: A hybrid approach for data warehouse view selection. Journal of Data Warehousing and Mining 2(2), 1–37 (2006)
13. Maxim, S., Spiros, M.: On the maintenance of UI-integrated Mashup Appliactions. In: International Conference on Software Maintenance (ICSM 2011), pp. 203–212 (2011)