Materialized View Selection Based on Adaptive Genetic Algorithm and Its Implementation with Apache Hive

Dongjin Yu [1], Wensheng Dou [1] Zhixiang Zhu [2] Jiaojiao Wang [1]

[1] College of Computer, Hangzhou Dianzi University,
Hangzhou, 310018, China, [*]
E-mail: yudj@hdu.edu.cn
[2] Bank of Ningbo,
Ningbo, 315100, China,
E-mail: zhuzx_hdu@163.com

Received 28 December 2014

Accepted 14 September 2015

Abstract

Frequently accessed views in data warehouses are usually materialized in order to accelerate the speed of querying big data. However, the view materialization itself incurs huge costs. Moreover, some latest products of non-traditional data warehouse software, such as Apache Hive, still lack the support of materialized views. In order to select the appropriate views to be materialized with the possible minimized cost, we propose a novel approach to the materialized view selection problem based on an adaptive genetic algorithm. We establish a cost model that integrates the query, maintenance and storage costs to evaluate the performance of approaches and measure the fitness of an individual in the genetic algorithm. In addition, we introduce the adjustable factors for crossover probability and mutation probability, allowing the genetic algorithm to run quickly and avoid premature convergence. We also conduct extensive experiments for its implementation with Apache Hive, which query and manage large datasets residing in distributed storage. Both the simulation results and experiments on Apache Hive show that the approximately optimal solution for selecting materialized views can be obtained effectively using the approach presented.

Keywords: materialized view, multi-dimensional lattice, genetic algorithm, cost model, adaptive, Apache Hive.

1. Introduction

Online analytical processing (OLAP) operations usually impose a lot of selection, projection, connection and aggregation computations upon the data warehouses. As the data sets increase in data warehouse day by day, the cost used for OLAP operations becomes extremely high [1]. One of the most effective ways to solve this problem is to build appropriate materialized views in the data warehouse [2]. Since the materialized views store pre-calculated aggregated information, it can answer the query directly without computation on the base tables, thus improving query efficiency [3].

However, because - usually - there are many aggregations that can be calculated, often only a predetermined number are fully calculated while the

[*]Xiasha Higher Education Zone, Hangzhou, Zhejiang, China, 310018.

remainders are solved on demand. The problem of deciding which aggregations (views) to calculate is known as the 'view selection problem', which can be constrained by the total size of the selected set of aggregations, the time to update them from changes in the base data, or both. The selection of the appropriate views to be materialized has proven to be a NP-hard problem [4]. Many approaches to the problem have been explored, including enumeration algorithms, greedy algorithms, genetic algorithms, etc. Although the adoption of enumeration algorithms could obviously lead to the optimal solution, they incur high costs. Unlike the other approaches, genetic algorithms - which are suitable for solving NP-hard problems - can obtain an approximately optimal solution of the problem after a finite number of iterations. However, the traditional approaches based on genetic algorithms usually are slow at searching the solutions and meanwhile are prone to premature convergence.

On the other hand, the size of data sets being collected and analyzed in the industry for business intelligence is growing rapidly nowadays, making traditional warehousing solutions prohibitively expensive. Hive, the open-source data warehousing solution built on top of Hadoop, facilitates querying and managing large datasets residing in distributed storage [5]. It is now widely used in companies like Facebook, and almost becomes the standard of petabyte scale data warehouse [6]. Unfortunately, While Hive is high-performance at complex data batch reading and analysis, it lacks efficient techniques for ad-hoc multidimensional query due to the absence of materialized views.

Therefore, in this paper, we present a novel approach to materialized view selection based on an adaptive genetic algorithm. Our approach can select the appropriate views to be materialized with less total cost. It improves on current genetic algorithms by using adaptive adjustment mechanisms that can accelerate the search speed of the genetic algorithm and avoid premature convergence. Its implementation with Apache Hive shows its effectiveness on the non-traditional distributed data storage based on Map/Reduce.

The rest of this paper is organized as follows. Section 2 describes the related principles and defines the materialized view selection problem. After Section 3 presents the cost model of materialized views that involves cost of query, maintenance and storage, Section 4 introduces in detail the adaptive genetic algorithm for solving the materialized view selection problem. Section 5 compares our approach with the traditional ones through a simulation case, followed by the experimental results of its implementation with Apache Hive in Section 6. Afterwards, Section 7 presents the related work. Finally, Section 8 concludes the paper and outlines the future work.

2. The Materialized View Selection Problem

2.1. Description of the Materialized View Selection Problem

Assume that there exist four tables in the data warehouse, i.e., *Sales*, *Parts*, *Customer* and *Supplier*, whose schemas are defined as follows:

Sales(*sp_id*, *p_id*, *s_id*, *c_id*, *sp_price*)
Part(*p_id*, *p_name*, *p_type*, *p_price*)
Customer(*c_id*, *c_name*, *c_age*, *c_address*)
Supplier(*s_id*, *s_name*, *s_address*, *s_telephone*)

Here, *Sales* is the fact table that consists of the measurements of the sales business, whereas *Part*, *Customer* and *Supplier* are its corresponding dimension tables that contain descriptive attributes of the fact table. For decision analysts, what they are interested in is the summary information of the *sp_price* measure on different dimensions. Without materialized views, it would take a large number of calculations - such as selection, projection and aggregation - to get a result for the same query each time. In contrast, once we have created the materialized views, the query result is cached as a concrete table that may be updated from the original base tables from time to time. Whenever a query addresses the *sp_price* measure, we convert it into queries against the underlying materialized tables. This enables much more efficient access, at the extra storage cost. We define the problem of materialized view selection as follows: *how are we to select the appropriate views to be materialized while minimizing the total costs of queries, maintenance and storage?*

2.2. *Multi-dimensional Lattice Model*

Different views are unlikely to have the same probability of being request in a query. Since Harinarayan et al. mentioned the extension of their basic model with some weights of different views [7], query frequencies have been commonplace in the optimization formulations of view selection problem. In order to accurately quantify the query, maintenance and storage costs of materialized views, we first build the multi-dimensional cube lattice according to the candidate views, and then choose the appropriate views to be materialized. More specifically, we introduce query frequency properties in the multi-dimensional cube lattice in order for frequently accessed views to be materialized with high priorities. For the sake of convenience, Table 1 presents the multi-dimensional lattices and their related group-by clauses for the example in the previous section. In addition, Tables 2 and 3 show the complete SQL query statements *Q1* and *Q2*, for the lattice nodes of *pc* and *p* respectively.

Table 1. Lattices and their corresponding group-by clauses

No.	Group-by clause	Lattices
1	Part.p_type, Supplier.s_address, Customer.c_address	psc
2	Part.p_type, Customer.c_address	pc
3	Part.p_type, Supplier.s_address	ps
4	Supplier.s_address, Customer.c _address	sc
5	Part.p_type	p
6	Supplier.s_address	s
7	Customer.c_address	c
8	Null	none

From the definitions of the complete query statements of *pc* and *p*, we know that if *pc*'s query results are materialized, the query results of *p* can be obtained from *pc*'s corresponding materialized view, rather than from base tables through the operations of selection, projection and aggregation, thus improving the query efficiency.

In order to build the multi-dimensional cube lattice, the queries with the same group-by clause can be resolved from the candidate view that corresponds to one of the multi-dimensional lattice nodes. Figure 1 illustrates the multi-dimensional cube lattice for the example indicated in Table 1, with each node marked with a serial number, the size of the view and the query frequency.

Table 2. The SQL statement *Q1* for Lattice *pc*

***Q1* definition**

SELECT Part.p_type, Customer.c_address, SUM(Sales.sp_price)
FROM Sales, Part, Customer
WHERE Sales.p_id = Part.p_id and Sales.c_id = Customer.c_id
GROUP BY Part.p_type, Customer.c_address

Table 3. The SQL statement *Q2* for Lattice *p*

***Q2* definition**

SELECT Part.p_type, SUM(Sales .sp_price)
FROM Sales,Part
WHERE Sales.p_id = Part.p_id
GROUP BY Part.p_type

In the multi-dimensional cube lattice, there is only one root node whose corresponding view must be materialized beforehand. The query results for other nodes can be gained by the materialized views of the root node or their *direc*t or *indirect parent nodes*. Here, considering a node in a cube lattice, the *direct parent nodes* refer to its directly connected nodes in its adjacent upper layer, whereas the *indirect parent nodes* refer to its indirectly connected nodes in all its upper nonadjacent layers. Taking the cube lattice in Figure 1 as an example, *ps* and *sc* are the *direct parent nodes* of *s*, whereas *psc* is the *indirect parent node* of *s*.

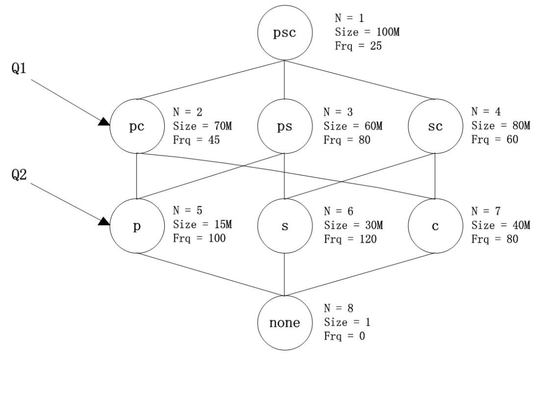

Fig. 1. A Multi-dimensional cube lattice

3. Cost Model of Materialized Views

Once the multi-dimensional cube lattice is established, we present its cost model in order to accurately quantify the query, maintenance and storage costs of materialized views in this section.

3.1. Query Cost

The query cost of the materialized views corresponding to the multi-dimensional cube lattice is defined as follows:

$$QueryCost(Q,M) = \sum_{q \in Q} f_q(q) qt(q,M) \tag{1}$$

in which, Q represents the candidate view set in the multi-dimensional lattice model, M represents the materialized view set, q represents a view query in Q, $f_q(q)$ represents the query frequency of q, and *qt(q, M)* represents *q's* query cost in the case where the materialized view set M already exists. The value of *qt(q, M)* can be obtained as follows:

$$qt(q,M) = \min_{k \in (F \cap M)} Size(k) \tag{2}$$

Here, F represents the set of q and its *direct* and *indirect parent* nodes in the multi-dimensional cube lattice. Besides, *Size(k)* represents k's query cost in the case where the materialized view set M already exists. In other words, we use the size of materialized view to denote the query cost on it.

3.2. Maintenance Cost

The maintenance cost of the materialized views corresponding to the multi-dimensional cube lattice is defined as follows:

$$\begin{aligned} maintenanceCost(Q,M) = \\ avg_{q \in Q} f_q(q) \sum_{v \in M} mt(v,M) \end{aligned} \tag{3}$$

Here, we introduce $avg_{q \in Q} f_q(q)$, or the mean value of the query frequency, to denote the maintenance frequency. Besides, *mt(v,M)* represents v's maintenance cost in the case where the materialized view set M already exists. The value of *mt(v,M)* can be obtained as follows:

$$mt(v,M) = \min_{k \in (H \cap M)} Size(k) \tag{4}$$

Here, H represents the set of v's *direct* and *indirect parent nodes*.

3.3. Storage Cost

The storage cost of the materialized views corresponding to the multi-dimensional cube lattice is defined as follows:

$$StorageCost(M) = \sum_{v \in M} Size(v) \tag{5}$$

in which, $Size(v)$ represents v's storage cost. In this paper, we use the physical storage size of v as the benchmark in measuring its storage cost.

3.4. Total Cost Measure Function

With the above cost definitions, we define the total cost of the materialized view as follows:

$$\begin{aligned} TotalCost(Q,M) = QueryCost(Q,M) + \\ \alpha(MaintenanceCost(Q,M) + \beta \cdot StorageCost(M)) \end{aligned} \tag{6}$$

Here, α and β represent the compensation factors, used to reasonably adjust the query, maintenance and storage costs as a proportion of the total cost. Because the value of storage cost is not in the same order of magnitude as that of query cost and maintenance cost, we set β as follows:

$$\beta = avg_{q \in Q} f_q(q) \tag{7}$$

Besides, we set α to 0.5, meaning that the query cost is more important than the maintenance and storage costs within the composition of the total cost. A possible value of α between 0.1 and 1.0 would be consistent with experience.

4. Selection of Materialized Views based on Adaptive Genetic Algorithm

A genetic algorithm (GA) is a search heuristic that mimics the process of natural selection [8]. The parallelism, self-study nature and robustness of the genetic algorithm make it very effective in solving the combinatorial optimization problem [9]. We apply an adaptive genetic algorithm to solve the materialized view selection problem. In particular, we employ the adaptive adjustment mechanism to accelerate the search speed and avoid premature convergence.

4.1. Encoding

Encoding transforms the feasible solution from the solution space of the problem to the searching space, in which the genetic algorithm can deal with it. For our approach, we use the binary coding to transform the candidate view set of the multi-dimensional cube lattice into an array of 0-1 integers. More specifically, the node's serial number corresponds to the array index, whereas the value of the element determines whether the view should be materialized. The array element with a value 1 means that its corresponding view needs to be materialized, whereas 0 means not. For the multi-dimensional cube lattice given by Figure 1, if the materialized view set M = 1, 3, 5, 6, 7, we use the following *coding array*:

1	0	1	0	1	1	1	0
1	2	3	4	5	6	7	8

4.2. Population Initialization

We use a random algorithm to generate n individuals as the initial population. For the given condition that the root node view in the multi-dimensional cube lattice must be materialized, it is necessary to assign 1 to the value of the individual's first code element if its initial value is 0.

4.3. Fitness Function

In a genetic algorithm, the greater the fitness of the individual, or the *coding array* in our approach, the greater the chance it goes into the next generation. During each successive generation, the individual with the greatest fitness in the population represents the locally optimal solution. We adopt the reciprocal of the materialized views' total cost as the value of the individual's fitness, which is defined as follows:

$$Fit(x) = \frac{1}{TotalCost(Q,M)_{CodingArray(M)=x}} \tag{8}$$

4.4. Selection Operator

As a process of screening, selection retains those individuals with high fitness and eliminates those individuals with low fitness within a population. The selection operator $p_s(x)$ that denotes the probability of an individual x evolving to the next generation, which can be calculated as follows:

$$p_s(x) = \frac{Fit(x)}{\sum\limits_{k \in Population} Fit(k)} \tag{9}$$

Here, *Population* represents a group of individuals and k represents an individual in that population.

4.5. Crossover Operator

To 'crossover' means to combine parent individuals according to certain rules and produce offspring individuals. It generates feasible solutions to the problem and searches the candidate solution at the same time. In our approach, we apply a two-point crossover as the crossover operator. An example of a two-point crossover is shown in Figure 2, in which we exchange the parent individuals' code segments separated by two randomly selected crossover points to generate the offspring individuals.

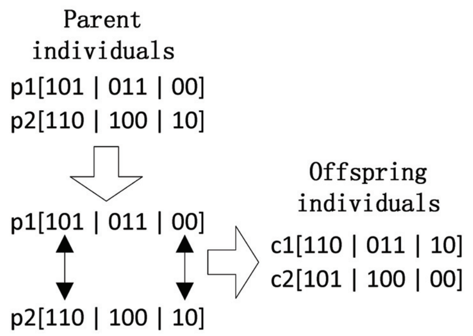

Fig. 2. An example of a two-point crossover

Compared with one-point crossover, two-point crossover can produce the more discrete offspring individuals, which leads to the higher likelihood of finding out the more optimized one.

4.6. Crossover Probability

The crossover probability p_c determines whether the two parent individuals perform a crossover operation. The larger the crossover probability, the faster the speed of the introduction of new individuals and the greater the loss rate of individuals with a high fitness. In contrast, if the crossover probability is set too small, it may cause a search block or premature convergence. Michael Negnevitsky, in [10], suggests that the crossover probability p_c be set to 0.7 for the best result. However, a fixed value of p_c may lead to a low search efficiency or premature convergence. Unlike traditional genetic algorithms, we set p_c between 0.4 and 0.99 and adjust dynamically according to the specific circumstances of the individual fitness. The probability p_c in our approach is defined as follows:

$$p_c = \begin{cases} 1 - \dfrac{1.2 \times Fit_{avg}}{Fit_{max} + Fit_{min}} & 0 < \dfrac{Fit_{avg}}{Fit_{max} + Fit_{min}} < 0.5 \\ 0.4 & else \end{cases} \tag{10}$$

in which, Fit_{max}, Fit_{min} and Fit_{avg} represent the maximum, minimum and average values of individual fitness in the population, respectively. As Formula (10) indicates, if more than half of the individuals have their fitness under the average value Fit_{avg}, the crossover probability will increase so as to introduce new offspring as often as possible and to improve the fitness of individuals in the population. Otherwise, if more than half of the individuals have their fitness above the average value Fit_{avg}, the crossover probability will decrease. In this way, it can effectively slow down the speed of losing individuals with high fitness and avoid premature convergence.

4.7. Mutation Operator

Mutation changes the values of some of the genes of individuals to increase the diversity of the population, thus alleviating premature convergence. Since the root view of the multi-dimensional lattice must be materialized, we only make changes to the gene elements of non-root views. We randomly select the non-root view corresponding to the gene element of the individual. If the value of a given gene element is 1, we change it to 0. Otherwise, if the value of a gene element is 0, we change it to 1.

4.8. Mutation Probability

The mutation probability p_m determines whether the parent individuals should perform the mutation operation. Michael Negnevitsky, in [10], suggests a small mutation probability p_m which ranges between 0.001-0.01 to reflect the little chance of mutation in the nature. We adopt the similar range and expand it a little for the more likelihood of achieving the approximately optimized result. In addition, on the point of convergence, increasing the value of the mutation probability p_m can avoid premature convergence. Otherwise, too large a mutation probability may degenerate the algorithm into a random search algorithm. Therefore, unlike traditional genetic algorithms, we define the mutation probability p_m as (11) indicates, so that it can be dynamically resized as needed.

$$p_m = \begin{cases} 0.1 & \dfrac{Fit_{avg}}{Fit_{max} + Fit_{min}} > 0.5 \\ 0.2 \times \dfrac{Fit_{avg}}{Fit_{max} + Fit_{min}} & else \end{cases} \tag{11}$$

4.9. Terminal Condition of Iteration

Theoretically, the genetic algorithm terminates only when it obtains the global optimal solution to the problem. However, this solution is usually unknown. Therefore, it is necessary to set some approximate convergence criteria to terminate the execution of the algorithm. For our approach, we preset a maximum number of iterations that are used as the termination condition.

4.10. Adaptive Genetic Algorithm Description

The process of an adaptive genetic algorithm (AGA) for selecting materialized views is shown in Table 4.

5. Simulation Case

In order to evaluate the correctness of the algorithm and the effectiveness of its adaptive adjustment mechanism, we carried out the simulation test on Windows 7 with an Intel Core 2 Duo E7500 (CPU 2.93 GHz, RAM 2GB). We compared our algorithm, namely AGA, with the IGA algorithm proposed in [11] and the QAGA algorithm proposed in [12] for solving the problem of selecting materialized views. The total cost of the materialized views is used as the criterion to evaluate all three algorithms. The IGA algorithm employs the improved traditional genetic algorithms for solving the materialized view selection problem. It increases the processing of invalid solutions to avoid the evolutionary stagnation. However, the improvement mechanism of the IGA algorithm is different from the adaptive adjustment mechanism proposed in this paper. Meanwhile, the QAGA algorithm considers the size and query frequency of the views and employs the greedy strategy to select top-k views, which can bring the maximum benefit if the views are materialized. The costs of the three algorithms for different dimension materialized view selection problems are shown in Table 5, in which the figures are the averages after running 1,000 times.

Table 4. The adaptive genetic algorithm for selecting materialized views

```
input:
sp_lattice: the multi-dimensional lattice
p_size: the population size
max_number: maximum number of iterations
output:
M: materialized view set

1 Begin
2    Initialize the population according to the
         encoding rules based on sp_lattice and p_size
3    g_number = 0
4    While (g_number < max_number)
5       Calculate the selection probability $p_s$
            according to Formula (9)
6       Do selection operation according to the
            selection probability $p_s$
7       Calculate the crossover probability $p_c$
            according to Formula (10)
8       Do crossover operation on the individuals
            according to $p_c$
9       Calculate the mutation probability $p_m$
            according to Formula (11)
10      Do mutation operation on the individuals
            according to $p_m$
11      g_number++
12   End While
13   Decode the result and output M
14 End
```

As Table 5 shows, no matter whether it deals with the low-dimensional materialized views or the high-dimensional ones, the AGA algorithm costs less than the QAGA algorithm. On the other hand, it has the same cost while solving low-dimensional materialized view selection problems as compared with the IGA algorithm, but a lower cost with the increasing scale of the problem. By comparing these experimental results, we can conclude that using the AGA algorithm can allow for the selection of the appropriate views to be materialized with less total cost.

Table 5. Comparison of the total costs of three algorithms

#D	AGA	IGA	QAGA
3	1,028,670	1,028,670	1,439,535
4	2,652,899	2,652,899	3,075,960
5	7,231,896	7,232,532	8,238,454
6	16,169,889	16,426,069	19,173,406
7	44,156,529	44,192,042	51,577,513
8	101,802,629	102,018,521	115,399,927

Note: #D represents the number of dimensions.

In order to further verify the effectiveness of the adaptive adjustment mechanism of the AGA algorithm, we employ the AGA algorithm and the GA algorithm to solve the eight-dimensional materialized view selection problem. The only difference between the AGA algorithm and the GA algorithm lies in the fact that the crossover probability p_c and mutation probability p_m of the GA algorithm cannot be changed once confirmed. The problem of eight-dimensional materialized view selection involves 256 candidate views. We set the population size to 100 and the number of iterations to 500. The iterative process of the GA algorithm and the AGA algorithm for solving the eight-dimensional materialized view selection problem is shown in Figure 3, in which the horizontal axis represents the number of iterations and the vertical axis represents the total cost of the materialized views.

As Figure 3 shows, the AGA algorithm nearly always has the fewer number of iterations than the GA algorithm does for either one-point crossover or two-point crossover, considering the same total cost. As we all know, a fewer number of iterations means less time, or fast speed of execution. Therefore, we can also draw the conclusion from Figure 3 that the search speed of AGA may be initially slower but later on always faster than that of GA. In other words, AGA can search steadily to obtain the approximately optimal solution. In addition, because AGA introduces an adaptive adjustment mechanism, it will speed up accordingly when it runs too slowly. Otherwise, it will slow down to prevent premature convergence. Figure 3 also shows that the two-point crossover achieves the better performance than the one-point crossover does.

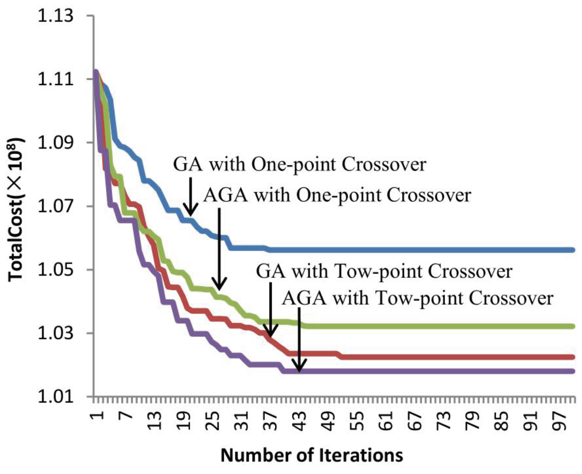

Fig. 3. The iterative processes of AGA and GA algorithms

6. Implementation with Apache Hive

The Apache Hive data warehouse software facilitates querying and managing large datasets residing in distributed storage. Considering the advantage of high efficiency, fault tolerance, and price-performance of Hadoop and Hive systems, they are frequently deployed as underlying platform for big data processing. However, in real business use cases, these data analysis applications typically involve multidimensional range queries [13]. While Hive is high-performance at complex data batch reading and analysis, it lacks the support of materialized view, one of the enablers of efficient multidimensional queries.

In order to make Hive process high volumes of data in an efficient way, we incorporated our approach into Apache Hive to support the view materialization. We constructed a five-node cluster, in which each node was configured with a four-core CPU of 2.66 GHz and 4GB memory. We used hadoop-1.2.1 as the underlying Map/Reduce and distributed storage framework, and hive-0.10.0 as the target data warehouse. The experimental data are the sales records of lotions from the well-known e-commerce companies, which can be downloaded from http://www.datatang.com/data/45764. In order to acquire the enough amounts, we simply extended the original 3-day records to one-month records by random. Table 6 shows the scheme and some sample sales records.

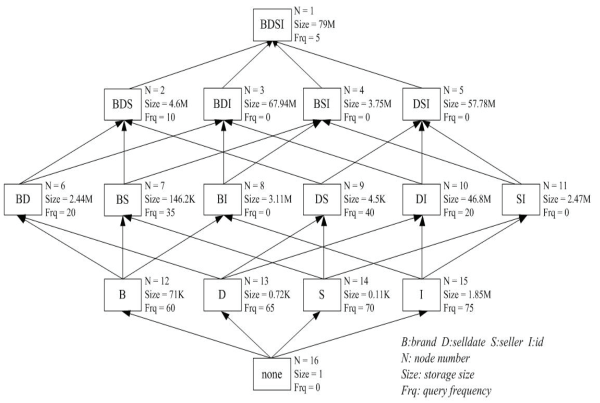

Fig. 4. The full multi-dimensional cube lattice for sales records

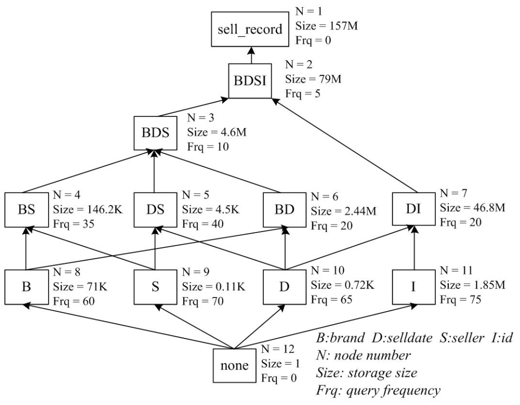

Fig. 5. The partial multi-dimensional cube lattice for sales records

During the experiment, we queried the sums of sales value with the groups of brands, selling date, seller, ID and the combination of above four. The experiment was continued for 5 rounds, each querying 80 times with the predefined grouping frequency. Figure 4 presents the full multi-dimensional lattice model, whereas Figure 5 just shows its partial one,

Table 6: Sales records: the scheme and the sample records

Brad	**Selling date**	**Seller**	**Price**	**Volume**	**ID**	**URL**
@nature	2013-10-20	Taobao	33.59	1	TB2469686929	htpp:// ···
@nature	2014-03-18	Jingdong	31.99	2	JD1298985434	htpp:// ···
······	······	······	······	······	······	······

which is closer to the real situations. The grouping frequencies, denoted by *Frq* in Figure 4 and 5 are given according to those in the real cases.

Figure 6 shows the time of 80 queries in seconds for the cases of *Not Materialized, All Materialized, Randomly Materialized and Materialized based on our approach (AGA)*.Since the root cuboid is employed to compute the results for the random non-group-by queries, it is always being materialized. In other words, not materialized here just refers to all non-root cuboids.

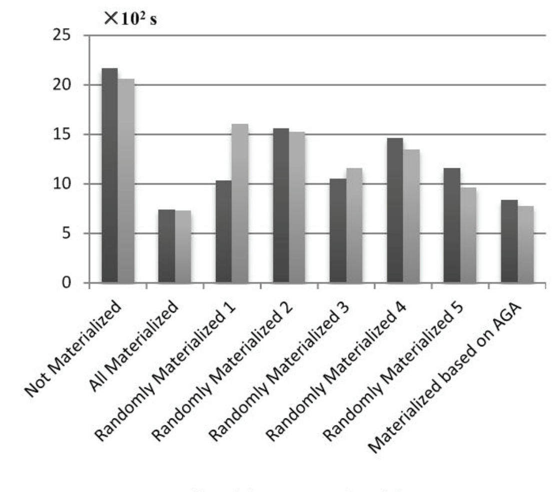

Fig. 6. The time in seconds of 80 queries for different materialized methods with Apache Hive

Figure 7 presents the cost for the cases of *Not Materialized, All Materialized, Randomly Materialized and Materialized based on our approach (AGA)*, using the full lattice model, whereas Figure 8 illustrates the cost using the partial lattice model. From both figures, we can find that our approach incurs the less total cost although its maintaining cost is not reduced considerably. However, the largest maintenance cost in Figure 8 is ascribed to *Random Materialized 3*, but not *All Materialized*. The reason is as follows: If multiple nodes have the same direct parent node which has not yet been materialized, maintaining these nodes would inevitably re-compute their indirect parent nodes for multiple times, thus increasing the total maintenance cost. In other words, the maintenance cost for *All Materialized* is not certainly the largest.

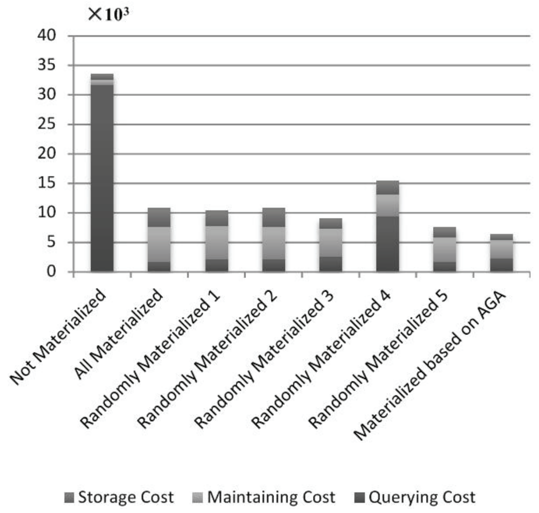

Fig. 7. The cost of different materialization methods with Apache Hive using full lattice model

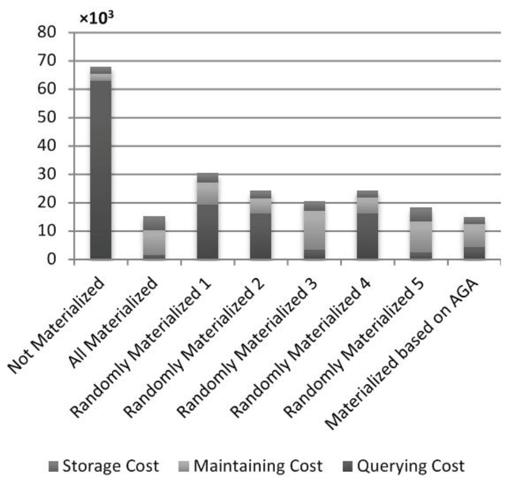

Fig. 8. The cost of different materialization methods with Apache Hive using partial lattice model

In order to further investigate how α in (6) impacts on our approach, we set α to different values between 0.1 and 1.0. As Figure 9 illustrates, the results of comparison are similar no matter how α is set, because the same cost model is employed. Here, a smaller α means we care much more about the query cost. An α of 0 simply means that we care about only the query cost, but no the maintenance and storage costs. Figure 9 also shows the query speed decreases with the growth of α, which is consistent with our cost model.

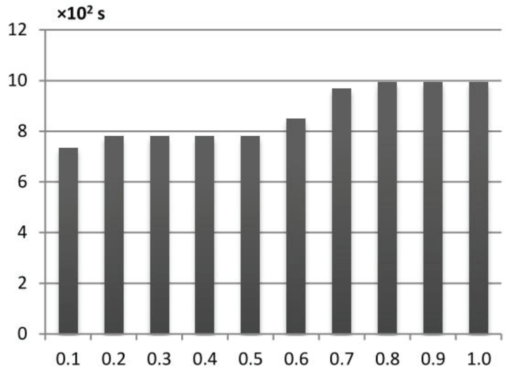

Fig. 9. The impact of α on the time of 80 queries in seconds on partial model with Apache Hive

7. Related Work

To make virtual views materialized enables much more efficient access, at the cost of extra data storage and some data being potentially out-of-date. It is most useful in data warehousing scenarios, where frequent queries of the actual base tables can be expensive. During the last decades, researchers have proposed many different approaches to the selection of the appropriate views to be materialized [14].

Venky Harinarayan et al. first define a data cube lattice of the materialized view selection problem [7]. They propose a greedy algorithm based on a data cube lattice which achieves good performance. Chuan Zhang et al. first utilize a genetic algorithm to solve the materialized view selection problem. They argue that the genetic algorithm is more practical and efficient than other heuristic algorithms [15]. In addition, Zhou et al. present an improved genetic algorithm, i.e., IGA, to solve the materialized view selection problem under query cost constraints [11]. IGA can avoid the evolutionary stagnation generated by invalid cycles, thus greatly improving the efficiency of the materialized view selection. However, it is unlikely to achieve the similar convergence results if repeating the evolving process for multiple times, because both the crossover probability and mutation probability of IGA are set too big. In addition, IGA does not employ the greedy strategy, which our approach does. Therefore, it is possible to have a decreased fitness during the evolving process and thus fails to converge to a steady result. The QAGA algorithm, proposed by Kumar and Haider in [12], takes the views query frequency into account. The materialized view obtained by the QAGA algorithm is said to have a higher query frequency in the practical situation. However, the QAGA algorithm does not consider the storage and maintenance costs. Besides, Xin Li et al. propose a novel Shuffled Frog Leaping (SFL) algorithm for materialized view selection [16]. Experimental results on the TPC-D benchmark data sets show that their proposed algorithm out-performs other well-known algorithms in terms of total maintenance costs and query response times. Badmaeva presents an algorithm on the basis of the data domain information for the materialized view selection problem [17]. Its advantage is in supporting the administrator by a proactive method to select the proper views to be materialized in the evolution process of the data warehouse.

As for the cost model of materialized views, there are also many researches presented in the literature. For instances, in [18], Talebian and Kareem introduce how to measure the query cost of the materialized views corresponding to the multi-dimensional lattice model. In [19], Lawrence presents the cost model for materialized view storage, while, in [20], Wang and Zhang discuss the storage cost. Most existing metric models, such as SCVSP [21], MCVSP [21] and MMVSP [1], include the constraints on storage space or update cost. Our proposed model however does not consider the constraints. Instead, we just include and try to minimize the storage space and update cost in addition to the query time in the objective function itself. We think that the introduction of constraints would bring about the difficulties or unfairness during the experiments for comparisons, simply because the actual constraints are arbitrary and thus really hard to be reasonably determined. Compared with UVSP, presented by Harinarayan et al. in [7], which simply minimizes the query and update time, our model considers the query cost, update cost and storage cost and integrates these costs on the same order of magnitude for a more comprehensive one.

Apache Hive is a widely used data warehouse system for Apache Hadoop, and has been adopted by many organizations for various big data analytics applications. In order to make Hive process increasingly high volumes data in a scalable and efficient way, Huai et al. aim to maximize the effective storage capacity and to accelerate data accesses to the data warehouse by updating the existing file formats [22]. They also improve cluster resource utilization and runtime performance of Hive by developing a highly optimized query planner and a highly efficient query execution engine. Although HiveQL offers similar features with SQL, it is still difficult to map complex SQL queries into HiveQL and manual translation often leads to poor performance. Xu et al. developed a tool named QMapper to address this problem by utilizing query rewriting rules and cost-based MapReduce flow evaluation on the basis of column statistics [23]. Indexing techniques are crucial for efficiency and scalability of processing queries over big data. In [24], Mofidpoor et al. propose an index-based join technique to speed up the process and integrate it in Hive by mapping their design to the conceptual optimization flow. To the best of our knowledge, however, the researches of Hive-based materialized views have not yet been carried out. In other words, we are the first to study the strategy of selecting views to be materialized for Apache Hive.

8. Conclusion

Evaluation results indicate that Apache Hive achieves acceptable performance for some data analysis tasks even compared with some high efficient distributed parallel databases. Nevertheless, it needs subtle adjustments of underlying storage facilities and indexing mechanism [25]. In this paper, we propose a novel approach to the selection of materialized views based on an adaptive genetic algorithm. It selects the appropriate views to be materialized for a locally minimized total cost. The experiment with Apache Hive verifies the appropriateness of the approach and the effectiveness of the adaptive adjustment mechanism. Its major contributions are as follows: (1) It establishes the metric model that integrates the total cost of the querying, maintenance and storage of materialized views, which can be used to measure the fitness of the individual in the adaptive genetic algorithm. (2) Different with our previous work in [26], it allows for the dynamic adjustment of the crossover probability and mutation probability, so as to accelerate the running speed and avoid premature convergence. (3) To the best of our knowledge, we present the first implementation of materialized views for Apache Hive in the literature.

Currently, we only focus on solving the problem of selecting materialized views effectively. However, for practical situations, keeping the materialized views updated is also a problem that needs to be solved. In the future, we will investigate approaches to the maintenance of existing materialized views at a minimal cost.

Acknowledgments

This work is supported by National Natural Science Foundation of China (No. 61472112), Zhejiang Provincial Natural Science Foundation (No. LY12F02003), Key Science and Technology Project Foundation of Zhejiang Province (No. 2012C11026-3) and Science and Technology Innovation Activity Plan for University Students of Zhejiang Province (No. 2014R407055). The authors would also like to thank anonymous reviewers and the colleagues in UC Santa Barbara who made valuable suggestions to improve the quality of the paper.

References

1. R. Hylock, F. Currim, "A maintenance centric approach to the view selection problem," *Information Systems*, (2013).
2. J. Li, X. Li, J. Lv, "Selecting Materialized Views Based on Top-k Query Algorithm for Lineage Tracing," *Proc. Third Global Congress on Intelligent Systems (GCIS)* (IEEE, 2012), pp. 46-49.

3. A. Cuzzocrea, M. S. Hacid, N. Grillo, "Effectively and efficiently selecting access control rules on materialized views over relational databases," *Proc. Fourteenth International Database Engineering & Applications Symposium*, (ACM, 2010), pp. 225-235.
4. J. S. Sohn, J. H. Yang, I. J. Chung, "Improved View Selection Algorithm in Data Warehouse," *IT Convergence and Security*, (Springer Netherlands, 2013), pp. 921-928.
5. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, Raghotham Murthy, "Hive - A petabyte scale data warehouse using hadoop," *Proc. 26th IEEE International Conference on Data Engineering*, ICDE. (2010), pp. 996-1005.
6. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, Raghotham Murthy, "Hive - A warehousing solution over a map-reduce framework," *Proc. the VLDB Endowment*, 2(2), (2009), pp. 1626-1629.
7. V. Harinarayan, A. Rajaraman, J. D. Ullman, "Implementing data cubes efficiently," *Proc. ACM SIGMOD Record*, 25(2), (ACM, 1996), pp. 205-216.
8. J. J. M. Mendes, J. F. Gonalves, M. G. C. Resende, "A random key based genetic algorithm for the resource constrained project scheduling problem," *Computers & Operations Research*, 36(1), (2009), pp. 92-109.
9. M. H. Mehta, "Hybrid Genetic Algorithm with PSO Effect for Combinatorial Optimization Problems," *International Journal of Advanced Computer Research*, 2(4), (2012), pp. 300-305.
10. Michael Negnevitsky, "Artificial Intelligence: A Guide to Intelligent Systems," Third Edition, Pearson Education Limited, (2011)
11. L. Zhou, X. He, K. Li, "An Improved Approach for Materialized View Selection Based on Genetic Algorithm," *Journal of Computers*, 7(7), (2012), pp. 1591-1598.
12. T. V. V. Kumar, M. Haider," A query answering greedy algorithm for selecting materialized views, Computational Collective Intelligence," *Technologies and Applications*, (Springer Berlin Heidelberg, 2010), pp. 153-162.
13. Yue Liu, Songlin Hu, Tilmann Rab, Wantao Liu, Hans Arno Jacobsen, Kaifeng Wu, Jian Chen, Jintao Li, "DGFIndex for smart grid: Enhancing hive with a cost-effective multidimensional range index," *Proc. the VLDB Endowment*, 7(13), (2014), pp. 1496-1507.
14. I. Mami, Z. Bellahsene, "A survey of view selection methods," *ACM SIGMOD Record*, 41(1), (2012), pp. 20-29.
15. C. Zhang, J. Yang, "Genetic algorithm for materialized view selection in data warehouse environments," *Data Warehousing and Knowledge Discovery*, (Springer Berlin Heidelberg, 1999), pp. 116-125.
16. X. Li, X. Qian, J. Jiang, et al., "Shuffled frog leaping algorithm for materialized views selection," *Proc. Second International Workshop on Education Technology and Computer Science (ETCS)*, (IEEE, 2010), pp. 7-10.
17. K. V. Badmaeva, "The algorithm of view selection for materializing in specialized data warehouses," *Proc. 34th International Convention of MIPRO*, (IEEE, 2011), pp. 1555-1559.
18. S. H. Talebian, S. A. Kareem, "A weight based genetic algorithm for selecting views," *Proc. 2012 International Conference on Graphic and Image Processing*, International Society for Optics and Photonics, (2013).
19. M. Lawrence, "Multiobjective genetic algorithms for materialized view selection in OLAP data warehouses," *Proc. the 8th annual conference on Genetic and evolutionary computation*, (ACM, 2006), pp. 699-706.
20. Z. Wang, D. Zhang, "Optimal genetic view selection algorithm under space constraint," *International Journal of Information Technology, 11(5), (2005)*, pp. 44-51.
21. H. Gupta, I.S. Mumick, "Selection of views to materialize in a data warehouse," *IEEE Transactions on Knowledge and Data Engineering, 17, (2005)*, pp. 2443.
22. Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N.Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, Xiaodong Zhang, "Major technical advancements in Apache Hive." *Proc. 2014 ACM SIGMOD International Conference on Management of Data*, (2014), pp. 1235-1246.
23. Yingzhong Xu, Songlin Hu, "QMapper: A tool for SQL optimization on hive using query rewriting," *Proc. 22nd International Conference on World Wide Web*, (2013), pp. 211-212.
24. Mahsa Mofidpoor, Nematollaah Shiri, T. Radhakrishnan, "Index-based join operations in Hive," *Proc. IEEE International Conference on Big Data*, (2013), pp. 26-33.
25. Taoying Liu, Jing Liu, Hong Liu, Wei Li, "A performance evaluation of Hive for scientific data management," *Proc. IEEE International Conference on Big Data*, (2013), pp. 39-46.
26. Zhixiang Zhu, Dongjin Yu, "Selecting Materialized Views based on Genetic Algorithm," *Advanced Science and Technology Letters*,Vol.45, (2014), pp.65-69.