Index Filtering and View Materialization in ROLAP Environment

Shi Guang Qiu
School of Computing
National University of Singapore
3 Science Drive 2 Singapore 117543
65-7918843

qiusg@singnet.com.sg

Tok Wang Ling
School of Computing
National University of Singapore
3 Science Drive 2 Singapore 117543
65-7722734

lingtw@comp.nus.edu.sg

ABSTRACT

Using materialized view to accelerate OLAP queries is one of the most common methods used in ROLAP systems. However, high storage and computation cost make this method very difficult to be implemented in the actual environment. Among various issues associated with this, index selection and view materialization are two of the top challenges. In this paper, we propose to build indexes on subsets of the primary keys rather than the full sets if the index selectivity for these smaller indexes can be maintained above the required level. Based on that we propose an index filtering rule, **Dominant Prime (DPrime) Index Set Filter**, to filter out candidate indexes that have insufficient index selectivity or have cheaper alternatives. In the second part, we propose a view materialization method, **Nested Relation Approach**, to group tuples with the same value for index attributes into one super tuple using a nested relation and implement this method using Oracle VARRAY. In performance tests, our method outperforms others significantly.

1. INTRODUCTION

In On-Line Analytical Processing (OLAP) systems, Multi-Dimensional model (MD model) and summary-views are most commonly used query optimization methods [KIM97], however storage cost and computation cost for summary-views and associated indexes increase explosively with the rising number of dimensional attributes [OLAP]. To implement these methods in the real environment is very difficult. In our previous paper [QL00], we have proposed two methods to filter out large number of unhelpful summary-views. Here, we would like to extend our research to view indexing and view reorganizing.

Indexes on summary-views are very important for OLAP query optimization, however they are also most expensive items in databases. Selection right index is very crucial for OLAP system design. First, we review attributes included in indexes. Because of the difference among dimensional attributes in OLAP environments, we might able to include only subsets of primary keys, defined as Dominant Primes or DPrimes, in indexes and get satisfied performance. Based on that, we develop an index filter, DPrime Index Set Filter to filter out those indexes, which have insufficient index selectivity or have cheaper alternatives. On top of that, we can still apply many existing index and view selection methods, e.g. [GHRU97], to further fine-tune the solution.

In addition, we propose our view materialization method, Nested Relation Approach, to optimize summary-views with non-unique indexes. By grouping few related tuples into one super tuple, we can "influence" the DBMS engine to pack these related data into fewer adjacent disk blocks, so I/O can be performed more efficiently. In addition, we can build indexes on groups of tuples rather than on each individual tuples, reduce the space required for indexes significantly. Moreover, we reduce the space requirement for summary-views. In the paper, we implement this method using Oracle Variable Array or VARRAY, a new feature introduced in widely adopted Oracle 8. The test result is very impressive.

In section 2, we discuss the background and related works. In section 3, a motivation example is given. In section 4 and section 5, we present our index filtering method and view materialization method. An experiment is shown in section 6 where our methods are compared with those of others. Finally, the conclusion is presented in section 7.

2. BACKGROUND and RELATED WORKS

A summary-view is grouping some measure attributes along various dimensions, i.e. corresponding to different sets of group-by attributes. By using summary-views, we can split OLAP query processes into few steps and perform costly aggregation operations in advance. To simplify the discussing below, we use SUM as the only aggregate function and assume that measure attributes are fixed. Thus, we can denote a summary-view as **SV(GA)** where GA is the set of dimensional attributes in the group-by clause, e.g. we denote the summary view below as SV(Product_Class).

```
SELECT    Product_Class, sum(Dollar_sold)
FROM      FACT_TABLE, PORDUCT_DIM
WHERE     FACT_TABLE.Product_Code =
          PRODUCT_DIM.Product_Code
GROUP BY  Product_Class
```


Conference '00, Month 1-2, 2000, City, State.

One of the most serious issues with summary-view method is the space explosion -- the space requirement increases explosively when more summary-views and indexes are added into OLAP databases [OLAP]. To overcome this problem, we have proposed two methods in our previous paper [QL00], Functional Dependency Filter and Size Filter, to filter out summary-views that are not very useful for OLAP systems. In this paper, we would like to further optimize OLAP systems by building indexes and reorganizing views.

There are a few proposals about selecting a set of views and indexes based on their contributions to a given set of queries. One of the most frequently referred papers is [GHRU97]. In this paper, a Greedy algorithm is proposed to treat indexes in a "similar" way as a view[1] and choose indexes for any selected view by greedily adding one index at a time, until the benefit per unit space of the view and the chosen indexes can no longer be increased. The paper claims that the result generated by this method is never worse than 47% of the optimal solution.

Also pointed out by [GHRU97], the number of possible index plan can be estimated to approximately 3n! for an n-dimensional data cube. An index filtering method is definitely required. However, in the paper, the index size is estimated to be the same size as the associated view. Which attributes and how many attributes are included in the index are not considered for index size estimation. Based on that, they developed their index filter and only different sequences of primary key are considered as index candidates. We believe this index filtering size estimation method is not accurate. For instance, Oracle B^+tree stores all values of index attributes in its leave node. The index size does related to sizes of attributes in the index. We believe it is possible to build indexes only on subsets of keys and provide sufficient query performance.

How to implement views and indexes is anther hot topic in OLAP area. There are many proposals about various kinds of indexes, e.g. Bitmap index, R-tree, Cubetree as well as their variations [OG95][OQ97][JS97][SAR97]. Some researches cover data materialization also [M98][KR98][RKR97]. The test results released in these papers are very impressive. However, most of these proposals are still in the research laboratories or depending on certain special software packages. In addition, there are also a few new developments from DBMS vendors and one of them is Oracle Index-Oriented table. By merging data into a B^+tree index, there are no needs to repeat data in a separate index and data access by full key is efficient. However, these B^+trees, which are holding all the information, could be very huge for many OLAP views and building, deleting, inserting on such big trees are very costly operations and OLAP system maintenance can be affected seriously.

3. MOTIVATION EXAMPLE

Example 1. The test system is built on NT PC running Oracle 8.16. The data and queries are generated by a program downloaded from the website of OLAP Council [OLAP]. Out of these data, only one star schema (Fig. 1) is used. The data statistics used below is collected in advance. Now, let us consider indexing and materialization issue for the fact table, SALES_FACT.

Solution 1. Index on primary key.

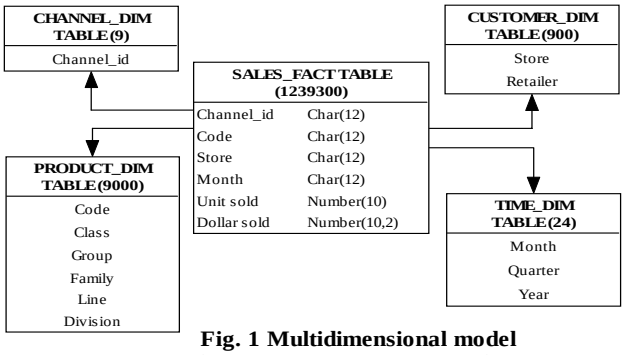

Fig. 1 Multidimensional model

(The number on the right of the table name indicates the number of tuples in that table)

The first solution is to materialize the fact table as a normal table and build an index on the primary key, (CHANNEL_ID, STORE,CODE,MONTH), as suggested by the Star Schema. The storage space required for the fact table and the index are 94MB and 112M. This index is very expensive.

Solution 2. Index on a subset of the primary key

The second solution is to replace the unique B^+tree on four dimensional attributes with a non-unique B^+tree index on only two dimensional attributes (STORE, CODE). As less attributes need to be stored in the new index, the index size shrinks by almost half to only 58MB. However, this saving is achieved at the cost of extra I/Os in the query processing. Based on the statistics below, on average 17 tuples will be fetched for each index search on this index. If the hardware is powerful enough, this index is still useful because of the significant saving for index storage.

$$\frac{\text{Size(SV(\{CHANNEL\_ID,STORE, CODE, MONTH\}))}}{\text{Size(SV(\{STORE, CODE\}))}} = \frac{1239300}{72900} = 17$$

(Where Size(V) is defined as a function which returns number of tuples inside V. All statistics used above are collected in advance.)

Solution 3. Nested Relation Approach.

The last solution is on top of the previous. In this plan, we reorganize the table as Nested_SALES_FACT and combine tuples with the same value for STORE, CODE into a big super tuple by keeping STORE and CODE as normal columns and grouping the rest of columns into a nested relation. In this test, we store this nested relation as a VARRAY attached to the super tuple[2]. The whole plan is shown in Fig. 2.

As tuples associated with each distinct pair of STORE and CODE can be fetched as one super tuple, query performance is drastically improved. Another big benefit is the significant reduction of the index size. As the number of tuples in

[1] Indexes cannot exist as standalone objects inside the database. Views must be selected before associated indexes.

[2] In case data are skew, we might need to pack tuples associated with the same STORE and CODE into few super tuples because of Oracle VARRAY implementation.

NESTED_SALES_FACT drops to 72900 from 1239300, down 94%, much less leave nodes are required in the index and space required is only 6MB or **10%** of the previous 58MB. The fact table shrinks significantly as well to only 47MB from 94MB, **50%** of the original size because of the removal of redundancy storage for STORE, CODE.

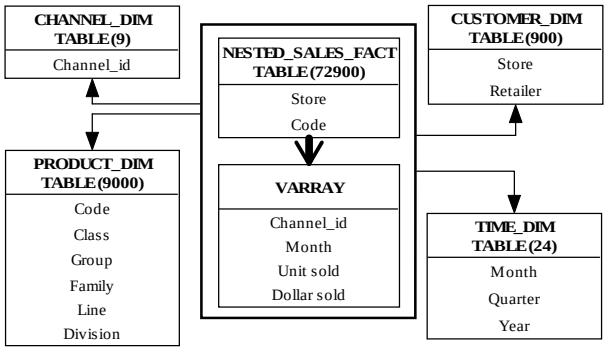

Fig. 2 Fact table organized with Oracle variable array
(with the number on the right of the table indicates the number of tuples in that table)

In the last part of this example, we randomly get 20 queries that are running against the fact table[3] from the query set generated by query generator from [OLAP] and run these queries against views prepared in above three solutions. The result is shown in Fig. 3. Solution 3 is significant better than the rest, for both storage cost and query performance.

	Solution 1. Index on primary key	Solution 2. Index on a subset of the primary key	Solution 3. Nested Relation Approach
Space requirement for tables	94MB	94MB	47MB
Space requirement for indexes	112MB	58MB	6MB
Total	206MB	152MB	53MB
Query Performance	0.302 sec/per query	3.943 sec/per query	0.236 sec/per query

Fig. 3 Performance Result

4. INDEX FILTERING

For index selection, the first major problem is the huge number of possible index plans. Putting aside index materialization, such kind of searching space is too big for many selection algorithms. Some kinds of index filtering algorithms are definitely required.

We start our discussion on index selectivity**,** the criteria for index selection. In an OLTP system, the index selectivity is defined as the average percentage of tuples associated with each instance of index attributes. The lower the percentage is, the lower percentage of tuples need to be retrieved from the view if this index is used in the query processing, i.e., the bigger I/O saving can be expected. An index is useful for a query processing unless its index selectivity is good[4]. In [O8TUN], the recommended value for the index selectivity is less than 2% to 4%.

While in an OLAP system, the most important query optimization goal is to push the query performance to the OLAP level, i.e. answering queries in seconds. To achieve that, we must control number of I/Os for each query. We believe "lower number of tuples", rather than "lower percentage of all tuples", need to be retrieved in a query is more important for good query performance. In this paper, we redefine the **INDEX SELECTIVITY** as the average number of tuples associated with each instance of index attributes. Comparing above two definitions of index selectivity, our definition is more emphasizing on real-time response. In the rest of paper, we will use our definition unless specified.

Example 2 Let's review the Example 1 posted in previous section, and look at following two cases of indexing on SALES_FACT table.

Case (1) If we have an index on attributes (STORE, CODE) for the SALES_FACT table, then the index selectivity for this index is 17 as shown in Example 1, while the selectivity in term of percentage is 0.00137%. As the index selectivity is quite good, we don't have to build some other bigger indexes such as (STORE,CODE,MOTH) if we already have this index.

Case (2) If we have an index on attribute STORE only for the view SALES_FACT table, then the index selectivity for it is 2420[5], while the selectivity in term of percentage is 0.19%. Obviously, this index is not very useful for OLAP queries as there are more than 2000 tuples need to be retrieved by the database engine.

$$\frac{\text{Size}(\text{SV}(\{\text{CHANNEL\_ID,STORE, CODE, MONTH}\}))}{\text{Size}(\text{SV}(\{\text{STORE}\}))} = \frac{1239300}{512} = 2420$$

(All statistics used above are collected in advance)

Form the above two cases, it is easy to observe that we can actually filter out lots of candidate indexes based on data statistics available. ■

In the next step, we would like to propose our heuristic index filtering rule to filter out indexes which are either not useful for OLAP queries or have cheaper alternatives. Before we dive into details, we would like to develop the notation for **INDEX SET**.

An **INDEX SET** X of SV(V) is a set of indexes of SV(V) on the same set of dimensional attributes X, regardless of attribute sequence, and denoted as $I_{SV(V)}(X)$.

In normal cases, different dimensional attributes are "playing" different roles inside each summary-view. Values of some dimensional attributes might repeat themselves less frequently in certain views and we can actually identify tuples in these views

[3] Queries are selected based on the attributes in the where clause and group-by clauses

[4] Issues with BITMAP indexes are not discussed here. In some cases, the queries do not access the table and purely work on the indexes

[5] Although there are 900 stores in the dimensional table, there are only 512 stores actually recorded in the fact table.

approximately by these attributes. Thus, we will call these attributes **"Dominant Prime**" or **DPrime** in short for these views.

Definition 1. Dominant Prime (DPrime). Assuming we have a summary-view SV(V) and X is a subset of a primary key. We define X is the DPrime for SV(V) iff the index selectivity for the index set X of SV(V) or size(SV(V))/size(SV(X)) is smaller than DOMRATE, a predefined constant. ■

The DOMRATE is a threshold for index selectivity, its value is depending on several factors, e.g. hardware, software and user requirement. The lower the DOMRATE is, the higher requirement for index selectivity. If the DOMRATE is 1, only the index sets on primary keys are selected. On the other end, the higher the DOMRATE, the higher requirement for hardware as more tuples need to be retrieved in the required time. In our paper, we determine the DOMRATE for our test system mainly on disk I/O and user requirement. Assuming the required system response time is 5 seconds. In Solution 2 of Example 1, we find that the system can handle test queries using an index with index selectivity of 17 in about 3 seconds. Therefore, we fix the DOMRATE as 20 conservatively.

Definition 3. Primary DPrime. In a MD model, X is a DPrime for SV(V), we define X as the Primary DPrime for SV(V), iff there is no DPrime Y for SV(V), which $Y \subset X$. ■

Similar to primary keys that we have defined in the traditional database, Primary DPrimes are smallest DPrimes that can satisfy the index selectivity requirement and are good candidates for indexes. It is quite straightforward if there is only one Primary DPrime for each view. However, in many cases, there could be more than one Primary DPrimes for a summary-view. Under such situation, we need to consider indexes on the union of Primary DPrimes as well. It is easy to prove that the union of Primary DPrimes is a DPrime also and the cost for the index on the union of DPrimes might be smaller than the combined cost for indexes on each individual DPrimes.

Index Filtering Rule — Dominate Prime (DPrime) Index Set Filter

For a summary-view, only index sets of its Primary DPrimes and their unions are considered as candidate index sets for the view■

In an MD model, the number of attributes holding tiny domains is quite big and index selectivity for the index set on them or few of them are normally not good enough. Thus, we can filter out a large number of candidate indexes based on DPrime Index Set Filter.

It is also true that the number of indexes selected by DPrime Index Set Filter might still be too big for materialization. We need to apply other algorithms based on other information, e.g., the Greedy algorithm [GHRU97] based on user queries. With this already greatly reduced searching space for indexes, these algorithms can be applied easily.

Example. 3. Following MD model (shown in Fig. 4) is from TPC-D, which is the same as the sample used in [GHRU97]. There are three attributes -- part, supplier, and customer and we abbreviate part to P, supplier to S and customer to C. The statistics for eight views are listed below.

Size(SV(PSC)) = 6 Million	Size(SV(PC)) = 6M
Size(SV(PS)) = 0.8M	Size(SV(SC)) = 6M
Size(SV(P)) = 0.2M	Size(SV(S)) = 0.01M
Size(SV(C)) = 0.1M	Size(SV(Ø)) = 1

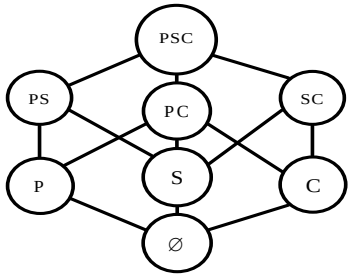

Fig. 4 TPC-D Example

In this example, we will compare our DPrime Index Set Filter with the index filtering algorithm used in [GHRU97] rather the Greedy index selection method proposed in the same paper. In fact, we need to use apply the same Greedy index selection method on top of the result of our DPRIM Index Set Filter**.**

Step 1. Index Filtering

DPrime Index Set Filter. We pick 20 as the DOMRATE and have:

- DPrime for SV(PSC) are PSC, PS, PC, SC with index selectivity of 1, 7.5, 1, 1.
- Primary DPrime for SV(PSC) are PS, PC and SC.
- The candidate index sets are $I_{SV(PSC)}(PS)$, $I_{SV(PSC)}(PC)$, $I_{SV(PSC)}(SC)$, $I_{SV(PSC)}(PSC)$

While for the index filter used in [GHRU97], only $I_{SV(PSC)}(PSC)$ is selected as the candidate set.

Step 2. Index Selection

In the sample of [GHRU97], the Greedy algorithm is used to select views and indexes based on a given set of user queries and three indexes on attributes (CSP), (PCS) and (SPC) are selected for SV(PSC). Based on the candidate index sets picked by our index filtering algorithm, indexes on (CS), (PC) and (SP) will be used instead.

Comparison between above two methods

Obviously, the storage saving for indexes is significant if our index plan is used. For each indexes, we only include two attributes instead of three attributes. For query performance, the index on (SP) might not be as good as the index on (SPC) for some queries, as the index selectivity for index on (SPC) is better. However, we believe the performance difference should not be significant as SP is still a DPrime. While comparing indexes on (CS) and (PC) with indexes on (CSP) and (PCS), our solution is better. With similar index selectivity, indexes picked by us require much less storage space. Thus, we believe our index filtering method is better that used in [GHRU97].

5. VIEW MATERIALIZATION

In a RDBMS system, information is logically manipulated as tuples. However, at lower levels, data are actually handled by much bigger units, e.g. blocks and pages. When a small tuple is fetched from harddisks, data stored nearby are also fetched in. It

is certainly much desirable if we can make full use of each I/O by packing tuples which are usually used in the same query processing together and reduce the total number of I/O required.

In query processing, data are accessed by either table scan or index search. With today's technology, it is still difficult to push the performance of table scan operations on huge tables to the OLAP level. In this paper, we will only discuss issues related to the index search. For an index search on an index, all tuples with the same value for index attributes are always fetched together. If the index selectivity is bigger, e.g. using DPrime rather than primary key, index search performance is certainly affected as shown in Solution 2 of the example 1. To solve this problem, we propose our view materialization method to reorganize views by grouping related tuples into a super tuple and building indexes on these super tuples instead.

View Materialization Method — Nested Relation Approach
For a summary-view with decided index plan, we define all dimensional attributes included in the index or indexes as **Index Attributes**, those not included as **Non-Index Attributes**; and reorganize the view by grouping tuples with the same values for Index Attributes into one super tuple and store all Non-Index Attributes in the original tuples as a nested relation attached to this super tuple. ■

Example 4. Let's review the fact table, SALES_FACT, used in Example 1 (as shown in Fig. 1). A non-key index on (Store, Code) is selected as the only index and its index selectivity is ρ = 17. Based on our Nested Relation Approach, we convert SALES_FACT to NESTED_SALES_FACT. (Fig. 5)

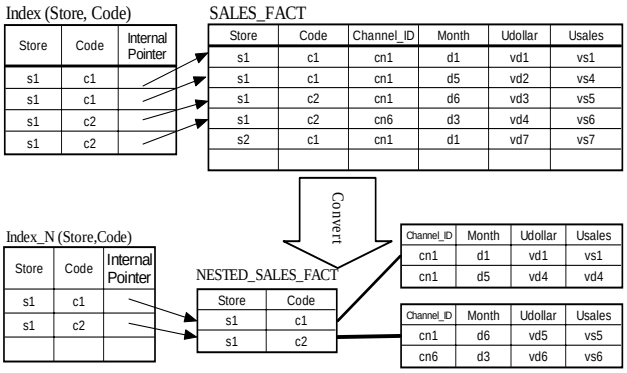

Fig. 5 Nested Relation Approach

The benefit of this proposal has already been shown in Case 3 of example 1.

- **Significant saving for index storage.** As tuples with the same values for index attributes will be grouped into a super tuple, the number of big super tuples is certainly much less, thus the size of the new index will be dropped significantly.

- **Better OLAP query performance.** By grouping related tuples together into nested relations and storing them physically together into few adjacent disk blocks, we can reduce the number of disk I/O access significantly for OLAP queries.

- **Significant saving for summary-view storage.** By grouping tuples together, index attributes with the same value only need to be stored once. The summary-view size can be reduced also. Thus, besides index search, we also accelerate table scan operation. ■

The first concern for this method is how the nested relation should be materialized. It is very difficult to do it in a traditional RDBMS. We can store the nested relation as a separate table to save some storage space or use cluster table for high performance. But all these methods have some serious weak points, e.g. high join cost, high storage overhead.

However, with the introduction of the new generation of ORDBMS, the database engine has greatly extended its functions. In Oracle 8, a new data type, VARRAY or variable array, has been added to database to support array and list operation inside the tuple. For the view shown in Example 4, view SALES_FACT can be materialized as Nested_SALES_FACT_V. (Fig. 6)

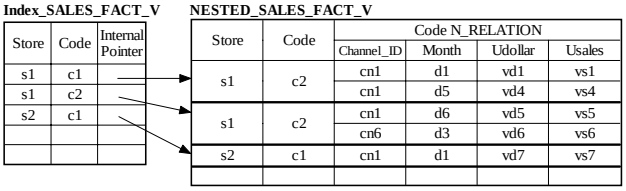

Fig. 6 Nested-Relation Approach (Oracle VARRAY)

Comparing with solutions in traditional RDBMS, VARRAY has some wonderful features very suitable for our Nested Relation Approach.

- **Low storage overhead.** The size of the array is of varying size. Although the maximum size for the VARRAY must be specified when it is declared, but the actual storage depends only on **the current count** of elements in the VARRAY. If data are skew, we can pack less tuples into one super tuple with limited storage overhead or pack more tuples into few super tuples. Although the index on these super tuples will not be a unique index, there is no significant impact for query performance if the B^+tree is small[6].

- **Good query performance**. The VARRAY requires **no joins** to retrieve data inside the nested relation. In addition, the VARRAY also gives better performance if the VARRAY is manipulated as a single unit in applications, e.g. OLAP batch loading.

- **Compatible with front-end tools.** In Oracle release 8.16, an additional feature has been added to allow VARRAYs to be viewed in the traditional flat (relational) form by using the TABLE syntax. For example 4, a normal view, VIEW_SALES_FACT, can be created as below to make the VARRAY object transparent to other applications. Many front-end tools can be used directly without any changes.

```
CREATE VIEW VIEW_SALES_FACT as
    SELECT  p.Store, p.Code, n.Channel_ID,
            n.Month, n.Udollar, n.Usales
    FROM    NESTED_SALES_FACT_V p,
```

[6] Base on our test, overhead for Oracle 8 is still quite high, but there are some significant improvements in Oracle 8i.

TABLE(p.N_RELATION) n

Because of the efficiency of VARRAY, we can pick a larger number for DOMRATE and try to pack more related tuples together if possible and further optimize OLAP systems. However if DOMRATE is too big, we might have too big super tuples for retrieval in query processing.

Certainly, Nested Relation Approach cannot be applied for all views, especially indexes with very low index selectivity (around 1). For views with multiple selected indexes selected, Nested Relation Approach might not be applied easily also. More research works are required in this area.

6. EXPERIMENT

Example 5. In this example, we continue the work in Example 1. Instead of working on only one view, we work on the whole MD model. At the first stage, we apply the Functional Dependency Filter and Size Filter [QL00] to filter out summary-views and get 33 views out of 4096 candidate views. For the performance test, we use Query Set 1 (Channel Sales Analysis) generated by the data generator program [OLAP]. These queries are randomly generated to simulate the ad hoc and dynamic nature of OLAP queries. There are 2500 queries in this set, but we run the first 500 queries only because of resource limitation. For each query, the test program built by us loads in query parameters and based on attributes in the group-by clause and where clause, rewrite the query so that queries can be redirected to the "fittest" summary-views, which are prepared in advance according to following materialization plans. Then the test program starts the query and logs the execution time.

Plan A. Flat Tables

All views are materialized as flat tables. For the biggest view, it will be materialized as:

A4095 (CODE, CLASS, GROUP, FAMILY, LINE, DIVISION, STORE, RETAILER, CHANNEL, MON, QTR, YR, USALES, UDOLLAR);

Space required these 33 tables in the Oracle database is about 1.07 GB. The average query processing time for these queries is 5.98 seconds.

Plan B. Star Schemas

All views are materialized as Star Schemas. For the biggest view, it is materialized as Fig. 1. Space required for these 33 fact tables inside these Star Schemas is about 658 MB and the space requirement for small dimensional tables (10 altogether) and associated indexes (very small) is additional 3 MB. Comparing with plan A, we have saved 40% of storage cost. However, if we use the traditional index strategy and build a B^+tree index on the primary key for each view, the storage space required for these 33 indexes is 690 MB and the total space requirement shoot up to 1.351 GB.

Again, we run the same 500 queries against this indexed view set. Average response time for these queries is only 0.16 seconds. It is a big improvement. Space requirement is still very huge and index loading and maintenance is very costly.

Plan C. Nested Relation Approach

First, we apply DPrime Index Set Filter on these 33 views. The DOMRATE we used in this example is 20 and get 52 primary DPrimes with four views with three primary DPrimes and eleven views with two primary DPrimes.

If we have a known set of user queries, we can apply the Greedy method used in [GHRU97] to pick various views and indexes. To be fair with another two plans, we select all these 33 views and select one index on primary DPrime for each view. For those views with multiple primary DPrimes, we will pick the smallest one. For the key sequence, we adopt a simple rule again and pick the attributes from product dimension followed by those from customer dimension, channel dimension and time dimension.

At the last step, we reorganize each view using the Nested Relation Approach and materialize all views using Oracle VARRAY. The space required for these 33 views is only 310 MB, or only 47% of space required for 33 views in Plan B. While these 33 indexes require only 33 MB, or 4.8% of 690 MB for 33 indexes in Plan B. The total space required for all dimension tables and indexes is 345MB, which is only 25% of overall space requirement for Plan B.

We get an even better result in the performance test. The average query processing time is 0.14 seconds. The summary of result for these three plans is shown in Fig. 7.

	Plan A Flat Tables	Plan B Star Schemas	Plan C Nested Relation Approach
Space requirement for tables	1070 MB	658 MB	310 MB
Space requirement for indexes	0	690 MB	33 MB
Space required for Dimension tables & indexes	0	3 MB	3 MB
Total	**1070 MB**	**1351 MB**	**346 MB**
Query Performance	**5.98 sec/per query**	**0.16 sec/per query**	**0.14 sec/per query**

Fig. 7 Performance Summary

The above results show that indexing is one of most efficient methods to boost the query performance in OLAP systems. However, the traditional index selecting method, indexing on keys, is not very suitable for OLAP systems. While with our newly proposed DPrime Index Set Filter and Nest Relation approach method, the space requirement for indexes and views reduces significantly. Moreover, the query performance gets boosted also.

7. CONCLUSION

In this paper, we propose the index filtering method**, DPrime Index Set Filter**, to filter out indexes that have insufficient index selectivity or have cheaper alternatives. In addition, we also propose a view materialization method, **Nested Relation Approach**, to reorganize views based on selected indexes using Oracle VARRAY. Besides accelerating OLAP queries efficiently, this method cuts down the storage cost significantly, especially for indexes.

In example 3, we apply our index filtering method on the same example used in [GHRU97]. While maintaining index selectivity above a pre-designed level, we build smaller indexes on subsets of the primary key. Comparison shows that our method greatly

reduces the storage cost for indexes and keeps sufficient query performance.

In addition, we also apply both of our methods upon a test system based on data from OLAP Council. Result shows our method is significant better than the widely used Star Schema. We achieve better query performance with 4.7% of storage space for indexes and 47% of storage space for tables, or less than 25% of total storage space required by Star Schema method. Although real systems are much bigger, we believe the efficiency of our methods should remain.

REFERENCES

[GHRU97] H.Gupta, V.Harinarayan, A.Rajaraman, and J.D.Ullman. Index selection for OLAP. In Proc. ICDE'97, pp. 208 – 219

[HRU96] V.Harinarayan, A.Rajaraman, J.D.Ullman, Implementing Data Cubes Efficiently, ACM SIGMOD '96, pp. 205-216

[JS97] T.Johnson and D.Shasha, Some approaches to Index Design for Cube Forests, IEEE Data Engineering Bulletin, MAR 1997, pp 36-35

[KIM97] R.Kimbal A Dimensional Modeling Manifesto AUG 1997 //www.dbmsmag.com

[KR98] Yannis Kotidis, Nick Roussopoulos, An Alternative Storage Organization for ROLAP Aggregate View Based on Cubetree, SIGMOD 1998

[M98] Guido Moerkotte, Small Materialized Aggregates: A light Weight Index Structure for Data Warehousing, Proc. VLDB'98

[O8TUN] Oracle 8 Turning, Release 8.0, pp. 10-3

[OG95] P.O'Neil, and G. Graefe, Multi-table joins through bitmapped join indices. SIGMOD'1995

[OLAP] //www.olapcouncil.org

[OQ97] P.O'Neil, D.Quass, Improved Query Performance with Variant Indexes, SIGMOD'97

[QL00] SG.Qiu, TW.Ling, View Selection in OLAP Environment, DEXA 2000

[RKR97] Nick Roussopoulos, Yannis Kotidis, Mema Roussopoulos, Cubetree: Organization of and Bulk Incremental Updates on the Data Cube, SIGMOD 1997

[SAR97] Sarawagi, Indexing OLAP Data, IEEE Data Engineering Bulletin, MAR 1997, pp 36-43