DYNAMIC PRE-FETCHING OF VIEWS BASED ON USER-ACCESS PATTERNS IN AN OLAP SYSTEM

Karthik Ramachandran, Biren Shah, Vijay Raghavan
P. O. Box 44330, CACS, University of Louisiana at Lafayette, Lafayette, LA 70504, USA

Keywords: data warehouses, OLAP, view selection, decision support systems, enterprise information systems

Abstract: Materialized view selection plays an important role in improving the efficiency of an OLAP system. To meet the changing user needs, many dynamic approaches have been proposed for solving the view selection problem. Most of these approaches use some form of caching to store frequently accessed views and a replacement policy to replace the infrequent ones. While some of these approaches use on-demand fetching, where the view is computed only when it is asked, a few others have used a pre-fetching strategy, where certain additional information is used to pre-fetch views that are likely to be accessed in the near future. In this paper, we propose a global pre-fetching scheme that uses user access pattern information to pre-fetch certain candidate views that could be used for efficient query processing within the specified user context. For specific kinds of query patterns, called drill-down analysis, which is typical of an OLAP system, our approach significantly improves the query performance by pre-fetching drill-down candidates that otherwise would have to be computed from the base fact table. We compare our approach against dynamat; a well-known on-demand fetching based dynamic view management system that is already known to outperform optimal static view selection. The comparison is based on the detailed cost savings ratio, used for quantifying the benefits of view selection against incoming queries. The experimental results show that our approach outperforms dynamat and thus, also the optimal static view selection.

1 INTRODUCTION

Decision Support Systems (DSS) involve complex queries on very large databases. While operational databases maintain state information, data warehouses typically maintain historical information. As a result, data warehouses tend to be very large and grow over time. To facilitate answering such complex queries that span over large amounts of data, the data is extracted, transformed and loaded into the warehouse and is stored in a way that supports common analytical operations. The data warehouse is generally organized as a set of fact tables and is indexed by attributes (primary keys) of the dimension tables that store dimension information. Fact tables are thin and long whereas dimension tables are thick and small. A star schema model, as shown in Figure 1 is used to represent a data warehouse.

In Figure 1, there are two dimension tables namely *Product* and *Location* and a central fact table *Sales*. The fact value that is measured is *sales*, which indicates the total sales of a particular product sold to a particular customer. In most real life applications, the dimensions are organized as hierarchies of attributes that are functionally dependent on the primary attributes of each dimension. A simple example is organizing the *Product* dimension into the hierarchy: *productId*, and *type*. A sample hierarchy for the schema of Figure 1 is shown in Figure 2.

While the data warehouse approach solves the problem of representing data in a form suitable for analytical queries, it does not completely address several other performance issues; for example, query response time for a given aggregated query. An Online Analytical Processing (OLAP) system consists of alternating query processing (i.e. when data warehouse is online) and maintenance windows (i.e. when data warehouse is offline). In a typical OLAP scenario, called drill-down analysis, a user successively asks queries that are more detailed. Roll-up is just the opposite. To improve query response times for such complex queries, intermediate results are materialized so that, when a query is asked, it can be answered from the already materialized results (if available). An obvious issue in using materialized results to answer queries is

Ramachandran K., Shah B. and Raghavan V. (2005).
DYNAMIC PRE-FETCHING OF VIEWS BASED ON USER-ACCESS PATTERNS IN AN OLAP SYSTEM.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 60-67
DOI: 10.5220/0002528700600067
Copyright © SciTePress

selecting views that should be pre-computed. The view selection problem has been shown to be an NP-Hard problem (Gupta Harinarayan and Rajaraman, 1997) and has been one of the major research issues. Several approaches have been proposed towards solving this problem. Some of the approaches (Gupta, 1997; Harinarayan, 1996; Shukla, 1998; Baralis, 1997; Bauer, 2003) suggest static selection of views before each query window and then using these pre-selected views to answer subsequent queries. The obvious drawback of this approach is that the selection algorithm needs to be run frequently enough to keep up with the changing user needs. Even under the assumption that query access patterns (see definition 3.4 for details) change only between successive query windows, the static approach is very inconvenient, since the algorithm has to be rerun after every query window and, as pointed out by (Shukla, 1998), this could take a long time.

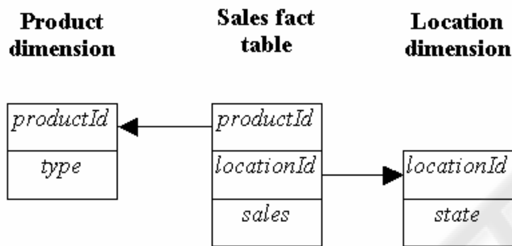


Figure 1: Sample Star Schema

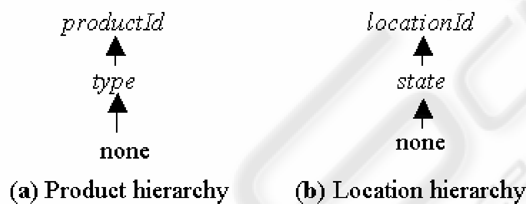


Figure 2: Dimension Hierarchy

To meet the changing user needs, several dynamic approaches (Kotidis, 2001; Sapia, 2000; Yao, 2003) have been proposed. These approaches work in a way similar to the principles of cache management in memories. The views may be fetched (or selected) on demand (on-demand fetching) or they may be pre-fetched using some prediction strategy. In cases where there are space constraints, a replacement algorithm may be used to identify the candidate victims for replacing the views in the materialized pool with new selections. The dynamic approach could be made to automatically adapt itself to changing query patterns.

The rest of the paper is organized as follows: Section 2 describes some of the related works in this field. In section 3, we introduce the lattice framework to model the dependency among views. Section 4 describes our proposed approach in more detail. In section 5, we provide experimental results of our work and compare it with (Kotidis, 2001). The last section summarizes, draws conclusions and presents the future work.

2 RELATED WORK

The Dynamat (Kotidis, 2001) approach implements dynamic view selection using the on-demand fetching strategy. The granularity of the materialized results is fixed to accommodate certain class of queries called Multidimensional Range Queries (MRQ). A MRQ is very similar to a view with the exception that the queries can also be single valued with respect to one of the dimensions (In OLAP context, these are called slice queries). The granularity of the MRQ is a compromise between choosing to materialize many small, highly specific queries and, materializing a few large queries and then answering incoming queries, at each stage, using them. Their approach, however, does not take the user access pattern information into account before making a selection.

The PROMISE (Sapia, 2000) approach goes one step further by predicting the structure and value of the next (incoming) query based on the current query. It argues that the number of possible queries is so large that predicting the next query as a whole (at a coarse granularity), is extremely time consuming. Instead, it requires that the granularity of the individual materialized results be detailed enough to capture the subtle differences between the values and structures of the addressed queries.

A different approach to view materialization is proposed in (Yao, 2003), where a set of batch queries are rewritten using certain canonical queries so that the total cost of execution can be reduced by using intermediate results for answering queries that appear later in the batch. Obviously this approach requires that all the queries must be precisely known before hand, and hence, even though the approach might work well in an operational database scenario, it might not be very useful in dynamic OLAP where it is extremely difficult to accurately predict the exact nature of queries.

In this paper, we propose a global pre-fetching scheme to pre-fetch certain candidate views that could be used for efficient query processing. Our scheme is global in the sense that the pre-fetched views are searched for in the *global* access lattice (for the particular user) at the beginning of a user query session or *context*. It emphasizes on the importance of the use of access pattern information to identify the user role so as to prune the access

space of the user accessing the lattice for selecting pre-fetching candidates. This information can be explicitly available by having a role associated with each user or it can be implicitly obtained by analyzing the query patterns of the user. For specific kinds of query patterns, called drill-down analysis, which is typical of an OLAP system, our approach significantly improves the query performance by pre-fetching drill-down candidates that otherwise would have to be computed from the base fact table.

3 LATTICE FRAMEWORK

OLAP views are typically represented as elements of lattices (Harinarayan, 1996) or 'View Graphs' (Gupta, 1997) to illustrate the fact that any view (except the top view, which is the base fact table) in the lattice can be computed by aggregating results from other (more detailed) views. For example, the schema of Figure 1 can be represented as a lattice as shown in Figure 3. Each node corresponds to a view in the multidimensional OLAP.

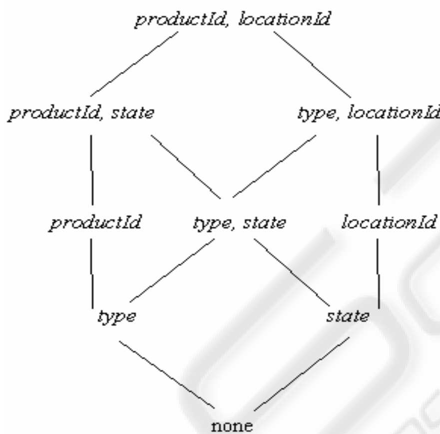


Figure 3: Lattice Cube

Definition 3.1 (*partial ordering*): Consider two views, v_1 and v_2. We say that $v_1 \prec v_2$ if and only if v_1 can be computed from v_2. We then say that v_1 is dependent on v_2. For example, in the lattice shown in Figure 3, the view (type) can be computed from (type, locationId). Thus (type) $\prec$ (type, locationId). There are certain views that are not comparable with each other using the $\prec$ operator. For example, (type) and (state) are not comparable with each other. Note that $\prec$ imposes a partial ordering on the views, and it is transitive. In order for a set of elements to be a lattice, any two elements must have a least upper bound and a greatest lower bound according to the $\prec$ ordering. However, in practice, we only need the assumptions that: (a) $\prec$ is a partial order, and, (b) there is a *top* element, a view upon which every view is dependent.

Definition 3.2 (*ancestors* and *descendents*): *ancestors* and *descendents* of a view v in the lattice are defined as follows:

$ancestor(v) = \{v' \mid v \prec v'\}$
$descendent(v) = \{v' \mid v' \prec v\}$

Definition 3.3 (*parents* and *children*): *parents* (*children*) of a view v in the lattice are defined as the immediate proper ancestors (descendents) of v. i.e.

$parent(v) = \{v' \mid v \prec v', \nexists x, v \prec x, x \prec v'\}$
$child(v) = \{ v' \mid v' \prec v, \nexists x, v' \prec x, x \prec v\}$

Definition 3.4 (*access pattern*): A query *access pattern* is defined as an ordered sequence of queries $q_1, q_2, \ldots, q_n$, addressed to views $v_1, v_2, \ldots, v_n$, respectively, such that $v_1 \prec v_2$, $v_2 \prec v_3$, …, $v_{n-1} \prec v_n$ i.e. given a query, a user successively drills-down the results for analytical processing.

4 PROPOSED APPROACH

The user access patterns play an important role in determining the granularity of materialization. For example, consider the scenario where a user, almost always, carries out a sequence of drill down operations on the lattice shown in Figure 3. Under such a situation, the MRQ level of granularity would be appropriate, since the user doesn't spend much time in accessing a single view. In other words, he is interested in only a slice of the view. Now consider a different situation where the user addresses a number of slice queries to the view (productId) before drilling down to the view (productId, state). If the granularity is kept at the MRQ level, then each slice query that the user addresses to the view (productId) needs to be fetched from the materialized parent and materializing the fetched result doesn't help in answering the future queries in any way. Instead, if the granularity were fixed at the view level, the whole view would be materialized when the first slice query is asked. The remaining slice queries addressed to the view can be answered directly from the materialized views. Thus, the access pattern information can be used to great advantage in determining the granularity and in pre-fetching views for materialization. Without loss of generality, we fix the granularity at the view level to simplify the explanation of the proposed approach.

There are two issues to be dealt with in the dynamic selection of views. The first issue deals with the amount of information that is used to select candidate views to be stored in the view pool (a view pool is a dedicated disk space for storing pre-computed views). At the most basic level, global

access frequencies can be used along with replacement strategies like least recently used, smallest penalty first, etc. for the selection of views. This approach, as used in (Kotidis, 2001), although very intuitive and simple, has some serious drawbacks. For example, in Figure 3, consider a scenario where the view (type, state) is seldom queried. Given that the most recent query asked by the user corresponds to the view (type) and that there isn't enough space available to materialize the results of this query; a replacement candidate, hence, needs to be determined. Under such circumstances, the (type, state) view would be a prime candidate for replacement due to its low access frequency. But suppose the past access patterns indicate that, although the view (type, state) has a low access frequency, the probability that it is queried given the previous query was addressed to the view (type), could be very high. In such a situation, it may not be appropriate to use replacement policies based on pure global frequencies or recentness of use while considering view benefits.

The second issue deals with the view fetching strategy, which can be either on-demand fetching (fetch on demand) or pre-fetching (fetch by prediction). On-demand fetching strategies are similar to the ones used in current operating system caches where pages are brought into memory only when requested. The advantage of this strategy is that the fetching algorithm need not be run unless the view corresponding to the query is absent from the view pool. However, the drawback of this approach is that for certain query patterns (drill-down queries), the performance may be very poor. For example, consider a query access pattern in the given order: (none), (type), (type, state), (productId, state), (productId, locationId). If pure on-demand fetching approach is used, the system will have to query the base fact table for every query in the pattern, since views are brought only on demand and no materialized view is available to answer the next query in the pattern. This is a typical OLAP scenario, called drill-down analysis, where a user progressively asks queries that are more detailed. In such a situation, even though it can be predicted (from past history) that the pattern may successively drill-down all through to the base fact table; the on-demand fetching strategy would continue to fetch views only when the queries are asked and hence the performance could become a bottleneck.

By pre-fetching certain views in advance, we can alleviate the above-mentioned drawbacks to a great extent. The most essential part of a pre-fetching strategy is a prediction algorithm that is able to predict the set of views that need to be brought in to the view pool.

4.1 Global Pre-Fetching Algorithm

We extend the Markov chain model (Howard, 1960) in developing a formal framework for modelling user interaction and navigation in an OLAP scenario. Each of the views visited (accessed) by the user map to the states of the Markov chain. The access pattern information that determines the probability that a user follows a particular navigation path can then be mapped to the state transition probabilities associated with a Markov chain. The probability that a user will drill-down to a particular view v_2 given that he is currently querying another view v_1 could be found by computing the transition probability between the two nodes in the nth degree probability matrix where n equals the number of hops (length of path) to reach state (view) v_2 from state (view) v_1. Drill-down analysis being the most natural (intuitive) way of querying an OLAP system, in our proposed approach, we do not explicitly emphasize roll-up queries since it can always be answered from the most recently materialized views.

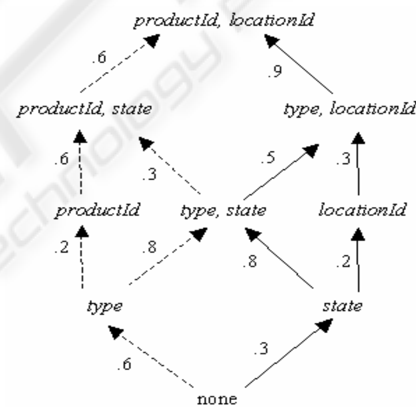


Figure 4: Lattice with Access Patterns and Roles

We now propose a global pre-fetching algorithm for dynamic view selection that pre-fetches candidate views based on the current query and the information about the users past access patterns. The algorithm is shown in Figure 5. The terminology used in the algorithm is shown below and explained in terms of the lattice in Figure 4 (modified lattice of Figure 3 with access patterns information).

q is a queue of vertices (nodes/views in lattice).

v is the current node.

w[] stores the weight (or benefit) of each of the nodes based on the probability of reaching them from the current node (the weight computation will be explained later).

eh[*i*][*j*] stores the edge probability for an edge connecting nodes *i* and *j*. It is the probability that the next query will be addressed to view *j* given the current query is addressed to view *i*. For example,

the value .2 (in Figure 4) along the edge connecting nodes (type) and (productId) is the probability that the next query will be addressed to view (productId) given the current query is addressed to view (type). The user access pattern information is used to determine the edge probabilities. These values are updated periodically, between successive query windows.

views is the set of all views/nodes in lattice.

TS is the total space available for storing pre-fetched views.

startNode is the node that represents the beginning of a query session or *context*.

prefetchedViews is the set of views returned by the pre-fetching algorithm.

size[*v*] denotes the size of the view *v*. We use our proposed method (Shah, 2004) for estimating the storage requirements of views, without actually materializing them.

```
Input: views, TS, startNode, size[]
Output: prefetchedViews
1.   add startNode to q;
2.   initialize w[startNode]=1, w[j (j≠startNode)]=0;
3.   while (! isEmpty(q))
     begin
4.      v = first element of q;
5.      for all a in parent(v)
        begin
6.        if (! stoppingCondition)
7.           add a to end of q;
8.        for each c in child(a)
9.           w[a] += eh[c][a] * w[c];
        end
     end
10.  sort descending views based on w[];
11.  k = 0, space = 0;
12.  while ((space + size[views(k)])< TS)
     begin
13.     prefetchedViews = prefetchedViews ∪ views(k);
14.     space += size[views(k)];
15.     k = k +1;
     end
16.  return prefetchedViews ;
```

Figure 5: Global Pre-Fetch Algorithm

The algorithm begins with the first user query (the start query). The algorithm carries out a breadth first search to compute the benefit (weight) for each of the ancestor views. According to this heuristic, the benefit of a view depends on the following two factors:

1. The number of descendant (all descendants including children) queries it can answer if materialized.
2. The probability that the user takes a path from the initial query (*startNode*) to the current view.

Note that the views that are projected as beneficial by the first factor above have many descendants, and as a result of which they form excellent candidates not only in supporting efficient drill-down analysis but also in (implicitly) facilitating roll-up queries.

The weight computation in line 9 in Figure 5 takes these factors into account. The multiplying factor *eh*[*c*][*a*], which is the probability associated with the edges, takes the second factor into account while the summation of this value over all the edges emanating from the node accounts for the first factor.

The stopping condition for the breadth first search can be fixed based on some threshold value that can be computed from the edge probability. In other words, if the edge probability falls below the threshold value, we stop pursuing nodes along that path. Once the weight values of nodes are computed, the best set of nodes can be selected for materialization depending on the available space. The threshold can be based upon one of the two different factors. The first factor is the maximum tolerable query response time. For example, as we go higher up the lattice, the cost of answering aggregated queries increases and hence the query response time increases. The second factor is the probability that the user queries a node given that he starts his analysis from *startNode*.

To illustrate weights computation, consider the lattice shown in Figure 4. Assume that the user starts his analysis from the *startNode* (none) and drills-down the lattice and that the stopping condition is to process all nodes up to a certain level (length of navigation path) with respect to the *startNode*. Assume that *level* = 2 and *w*[*startNode*] = 1.

when *level*=1
w[type]=.6*1=.6, *w*[state]=.3*1=.3
when *level*=2
w[productId]=.2**w*[type]=.12
w[type,state]=.8**w*[type]+.8**w*[state]=.72
w[locationId]=.2**w*[state]=.06

Since the stopping condition (*stoppingCondition*) is *level* = 2, using above values, the node (type, state) would be a good candidate for pre-fetch since it has the maximum weight.

The materialized views can be used to answer user queries as long as the user stays in the same context. The context is defined by the past access patterns of the user. As long as the user navigates within the same context, no (or minimal) fetching is required after the initial fetch. However, to avoid intermittent delays, materialization of selected views could be done in *parallel* with query processing.

4.2 Lattice Pruning

The number of views examined by the global pre-fetch algorithm can be pruned by taking into account the access pattern information and the user role. In a typical OLAP scenario, most of the time, a user is confined to a region of the lattice that interests him based on his profile or role. The user has different roles depending on the start node and the navigation path pursued. For example, the lattice of Figure 4 has two dimensions, namely product and location. A product manager drills-down through a region of the lattice that groups the facts by the product dimension (shown by dotted arrows) whereas; a regional manager drills-down a region of the lattice that groups the facts by the location dimension (shown by solid arrows). Even though the actual nodes accessed by the user could vary, the region of the lattice is more or less determined by the specific role of the user. By knowing the role of the user, one can prune the lattice space to search for nodes relevant to the current role.

5 EXPERIMENTS

A detailed set of experiments were carried out to measure the effectiveness of our proposed global pre-fetching scheme against a well-known dynamic view management system (dynamat) (Kotidis, 2001) that uses on-demand fetching strategy.

Table 1: Schema 1

Dimension Number	Dimension	Hierarchy	
		1	2
0	1000	200	50
1	1000	500	100

Table 2: Schema 2

Dimension Number	Dimension	Hierarchy		
		1	2	3
0	100	50	10	2
1	75	25	5	-
2	50	25	10	-
3	25	5	-	-

Synthetic data sets were used for generating multidimensional data. Table 1 and Table 2 contain the schemas and the number of distinct values of the dimensions and hierarchies of the two synthetic databases (schema 1 and schema 2) that we used. For example, the data in Table 1 means that the schema 1 has two dimensions. Dimensions 0 and 1 have a two-level hierarchy. Both dimensions have 1000 distinct values. Dimension 0 hierarchies have 200 and 50 values respectively, while dimension 1 hierarchies have 500 and 100 values respectively. The total number of views for schema 1 and schema 2 are 16 and 240, respectively. The maximum size of the base fact table is 1 million tuples for schema 1 and 9.375 million tuples for schema 2. For experimental purposes, a data density of 1% for schema 1 (approx. 10,000 tuples) and 10% for schema 2 (approx. 937,500 tuples) is selected and the total size of all the views in the multidimensional data cube is approximately 100,000 tuples for schema 1 and 5.6 million tuples for schema 2.

5.1 Performance Evaluation

To compare the two approaches, we measure the following:

1. The cost of answering the query from the matching view. This is assumed to be equal to the number of tuples (size) in the view. The cost is measured using the Detailed Cost Savings Ratio (DCSR) (Kotidis, 2001). If c_i is the cost of execution of query q_i from the base fact table, c_v is the cost of execution of q_i from the matching view v and M is the set of materialized views in the view pool then,

$$DCSR = \frac{\sum_i s_i}{\sum_i c_i}, where$$

$$s_i = \begin{cases} 0 & \text{if } q_i \text{ cannot be answered from } M \\ c_i & \text{if there is an exact match for } q_i \text{ in } M \\ c_i - c_v & \text{if } v \text{ from } M \text{ was used to answer } q_i \end{cases}$$

   Thus, to maximize the overall performance, DCSR values should be as high as possible.
2. Given a space constraint, the total number of view replacements or Cumulative Replacement Count (CRC) in the materialized pool with new selections.

5.2 Generating Query Patterns

To compare our approach against the dynamat approach, we generated a set of query patterns (for drill-down analysis) that are representative of OLAP queries. The access information was embedded into the lattice by arbitrarily assigning probabilities between 0 and 1 to all edges emanating from each of the nodes (ensuring that the sum is never greater than 1). While generating the patterns, there are some issues that need to be taken into account. Given that a user is currently querying a view v_i, the next view v_j in the access pattern is chosen based on the emanating edge probabilities. For this purpose, we used the Roulette Wheel Selection strategy,

which randomly picks objects based on their assigned weights.

For testing purposes, we generated a set of 10 query patterns for schema 1 each consisting of 3 queries and a set of 50 query patterns for schema 2 each consisting of 9 queries. The patterns were generated by randomly choosing a node as start node and then generating the sequence of queries from the start node. Each new pattern denotes a change in the context. Our approach is affected by the context change, since its selection is based on views that are best suited for the current context. Dynamat, however, is not affected by the context change since it does not exploit the user access patterns.

5.3 Results

Performance was measured under different space constraints (i.e. view pool size expressed as a percentage of the full data cube size). The DCSR per view (in decreasing order of savings) for schema 1 and schema 2 (for space constraints of 5%, 10% and 20%) are shown in Figure 6 and Figure 7, respectively. The CRC for schema 1 and schema 2 are shown in Figure 8 and Figure 9, respectively. The global pre-fetching scheme clearly outperforms the dynamat approach, especially when the available space is low. As the available space increases, the query performance (DCSR) of dynamat gradually approaches to that of ours. Dynamat chooses views for materialization as and when new queries are asked. Our pre-fetching approach selects views for materialization at the beginning of every context. With more available space, more views can be materialized, as a result of which the probability of finding a matching view to answer a query is high. Additionally, the global pre-fetching scheme uses the access patterns information, which further optimizes the selection of views in any given context, as seen by the high DCSR values. On the other hand, when the space constraints are high, dynamat, which updates its selection at each stage, requires replacing a lot of views. In the process, the DCSR per view drops since more views have to be answered from the base fact table. The global pre-fetch, however, continues to perform better since it selects views at the beginning of every context and the selection is such that the queries in the given context are likely to be answered from the materialized view pool, instead of the base fact table. Additionally, the global pre-fetch requires fewer number of replacements since many of the selections persist over different contexts, as a result of which the reusability of these already materialized views for answering queries from other contexts increases.

It has been experimentally proved in (Kotidis, 2001) that dynamat outperforms the optimal static view selection. The results above show that our approach outperforms dynamat and thus, also the optimal static view selection.

6 CONCLUSIONS

Pre-computation of views is an essential query optimization strategy for decision support systems. To meet the changing user needs, the views may be fetched (or selected) on demand (on-demand fetching) or they may be pre-fetched using some prediction strategy. In this paper, we proposed a global pre-fetching scheme that uses user access pattern information to pre-fetch certain candidate views that could be used for efficient query processing within the specified user context. Our approach optimizes the selection of views for efficient drill-down analysis, which is the most natural way of querying an OLAP system. Roll-up analysis is not explicitly emphasized since such queries can always be answered from the most recently materialized views.

We compare our scheme against dynamat, a dynamic view management system that uses on-demand fetching and is already known to outperform optimal static view collection. The DCSR results show that the average cost savings of answering a query using our proposed scheme clearly exceeds the dynamat approach. The CRC results show that our scheme is more robust than dynamat since it requires relatively fewer number of view replacements.

In future, we plan to test our approach by varying the granularity of the materialized results and also on large real-world data sets.

REFERENCES

Baralis, E., Paraboschi, S., Teniente, E., 1997. Materialized View Selection in a Multidimensional Database. In *Proc of 23rd VLDB Conf.*, pp. 156-165.

Bauer, A., Lehner, W., 2003. On Solving the View Selection Problem in Distributed Data Warehouse Architectures, In *Proc. of SSDBM Conf.,* pp. 43-51.

Harinarayan, V., Rajaraman, A., Ullman, J., 1996. Implementing Data Cubes Efficiently. In *ACM SIGMOD Conference*, pp. 205-216.

Gupta, H., 1997. Selection of Views to Materialize in a Data Warehouse. In *Proc. of Intl. Conf. on DB Theory*, pp. 98-112.

Gupta, H., Harinarayan, V., Rajaraman, A., 1997. Index Selection for OLAP. In *13th Conf. on Data Engg*, pp. 208-219.

Howard, R., 1960. Dynamic Programming and Markov Processes. *MIT Press*.

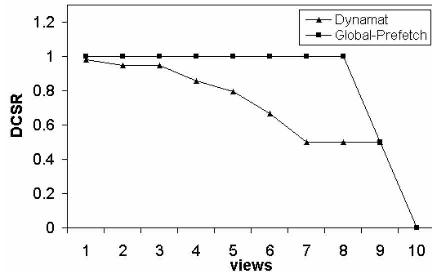
Kotidis, Y., Roussopoulos, N., 2001. A Case for Dynamic View Management. In *ACM Transactions on Database Systems*, vol. 26, no. 4, pp. 388-423.

Sapia, C., 2000. PROMISE: Predicting Query Behaviour to Enable Predictive Caching Strategies for OLAP Systems. In *Proc. of Intl. Conf. on Data Warehousing and Knowledge Discovery*, pp. 224-233.

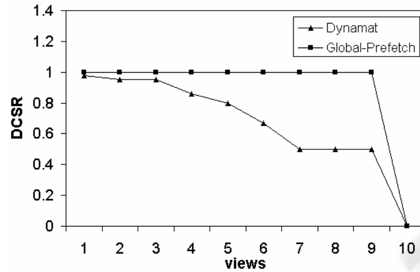
Shah, B., Ramachandran, K., Raghavan, V., 2004. Storage Estimation of Multidimensional Aggregates in a Datawarehouse Environment. In *Intl. Conf. on Systemics, Cybernetics and Informatics*, pp. 283-290.

Shukla, A., Naughton, J., Deshpande, P., 1998. Materialized View Selection for Multidimensional Datasets. In *Proc. of 24th VLDB Conf.*, pp. 488-499.

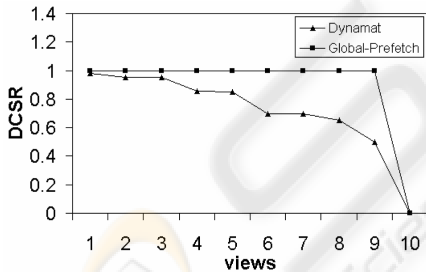
Yao, O., An, A., 2003. Using User Access Patterns for Semantic Query Caching. In *Intl. Conf. on Database and Expert System Applications*, pp. 737-746.



(a) space = 5%

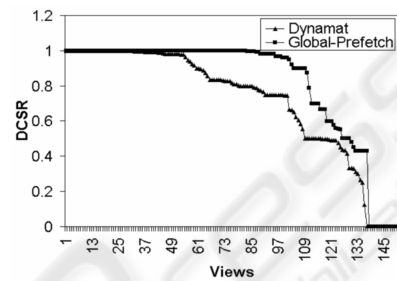


(b) space = 10%

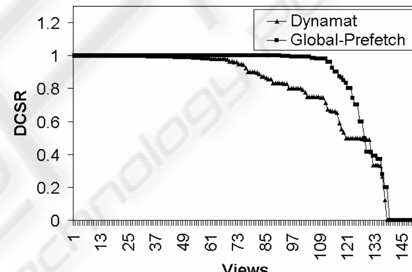


(c) space = 20%

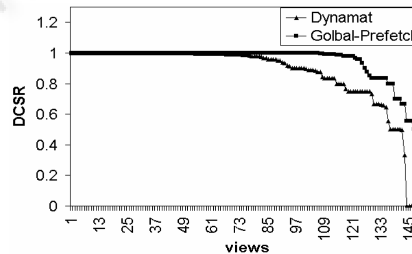
Figure 6: DCSR per view (Schema 1)



(a) space = 5%



(b) space = 10%



(c) space = 20%

Figure 7: DCSR per view (Schema 2)

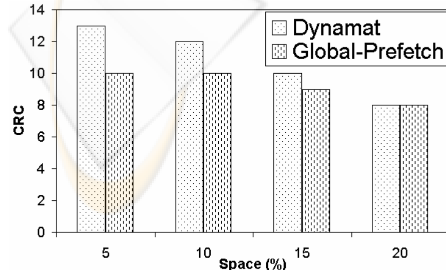


Figure 8: CRC (Schema 1)

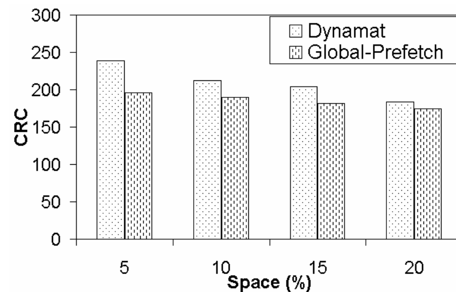


Figure 9: CRC (Schema 2)