Aggregate Aware Caching for Multi-Dimensional Queries

Prasad M. Deshpande[*] and **Jeffrey F. Naughton**

University of Wisconsin, Madison, WI 53706
{pmd, naughton}@cs.wisc.edu

Abstract. To date, work on caching for OLAP workloads has focussed on using cached results from a previous query as the answer to another query. This strategy is effective when the query stream exhibits a high degree of locality. It unfortunately misses the dramatic performance improvements obtainable when the answer to a query, while not immediately available in the cache, can be computed from data in the cache. In this paper, we consider the common subcase of answering queries by aggregating data in the cache. In order to use aggregation in the cache, one must solve two subproblems: (1) determining when it is possible to answer a query by aggregating data in the cache, and (2) determining the fastest path for this aggregation, since there can be many. We present two strategies – a naive one and a *Virtual Count* based strategy. The virtual count based method finds if a query is computable from the cache almost instantaneously, with a small overhead of maintaining the summary state of the cache. The algorithm also maintains cost-based information that can be used to figure out the best possible option for computing a query result from the cache. Experiments with our implementation show that aggregation in the cache leads to substantial performance improvement. The virtual count based methods further improve the performance compared to the naive approaches, in terms of cache lookup and aggregation times.

1 Introduction

On-Line Analytical Processing (OLAP) systems provide tools for analysis of multi-dimensional data. Most of the queries are complex, requiring the aggregation of large amounts of data. However, decision support applications need to be interactive and demand fast response times. Different techniques to speed up a query have been studied and implemented, both in research and industrial systems. These include precomputation of aggregates in the database, having specialized index structures, and caching in the middle tier.

While a great deal of work has been published on these issues, to our knowledge the published literature has not addressed the important issue of building an "active cache", one that can not only speed queries that "match" data in the cache, but can also answer queries that require aggregation of data in the cache. We show that a cache with such an ability is much more effective than a cache without such a capability. Intuitively, this is straightforward: aggregating cache-resident data is much faster than issuing a SQL query to a remote data source. However, in practice making this work is non-trivial.

The first issue to be dealt with is that in such an active cache the lookup process is considerably more complex than it is in an ordinary cache, because it

[*] Work done while at NCR Corp.

is not sufficient to see if the query result is in the cache. One must determine if the data in the cache is a sufficient basis from which to compute the answer to the query. This problem is especially difficult with fine granularity caching schemes such as chunk-based caching [DRSN98], query caching [SDJL96], and semantic caching [DFJST]. Obviously, this lookup must be fast — it is infeasible to spend a substantial amount of time deciding if a query can be computed from the cache, because it is possible that the lookup cost itself could exceed the time required to bypass the cache and execute the query at the remote backend database.

The second issue to be dealt with is that in such an active cache, there can be multiple ways in which to perform the aggregation required to answer the query. This situation arises due to the hierarchical nature of OLAP multidimensional data models — in general there are multiple aggregation paths for any query. The multiple aggregation paths complicate the cache lookup problem even further, since now not only is it necessary to determine if a query is computable from the cache, one must also find the best way of doing this computation.

In this paper, we propose solutions to both the cache lookup problem and the optimal aggregation path problem. Our implementation shows that with a small space overhead, we can perform much better than naive approaches. Overall, an active cache substantially outperforms a conventional cache (which does not use aggregation) for the representative OLAP workloads we studied.

Related Work In the field of caching for OLAP applications, [SSV] presents replacement and admission schemes specific to warehousing. The problem of answering queries with aggregation using views has been studied extensively in [SDJL96]. Semantic query caching for client-server systems has been studied in [DFJST]. [SLCJ98] presents a method for dynamically assembling views based on granular view elements which form the building blocks. Another kind of caching is chunk-based caching which is a semantic caching method optimized for the OLAP domain. Our previous paper [DRSN98] describes chunk based caching in detail. The different methods of implementing aggregations compared in this paper are based on a chunk caching scheme. A recent work on semantic caching is based on caching Multidimensional Range Fragments (MRFs), which correspond to semantic regions having a specific shape [KR99].

Paper Organization The remainder of the paper is organized as follows: Section 2 gives a brief description of the chunk based scheme; Section 3 presents an exhaustive search method for implementing aggregations; and Section 4 describes the virtual count based strategy. These methods are extended to incorporate costs in Section 5. Section 6 discusses replacement policies and Section 7 describes our experiments and results. The conclusions are presented in Section 8. Some details have been skipped in this paper due to space limitations. More details are available in [D99].

2 Chunk based caching

Chunk-based caching was proposed in [DRSN98]. In this section, we review chunk-based caching in order to make this paper self-contained. Chunk based caching takes advantage of the multi-dimensional nature of OLAP data. The dimensions form a multi-dimensional space and data values are points in that space. The distinct values for each dimension are divided into ranges, thus dividing the multi-dimensional space into chunks. Figure 1 shows a multidimensional

space formed by two dimensions $Product$ and $Time$ and the chunks at levels $(Product, Time)$ and $(Time)$. The caching scheme uses chunks as a unit of caching. This works well since chunks capture the notion of semantic regions. Note that there can be chunks at any level of aggregation.

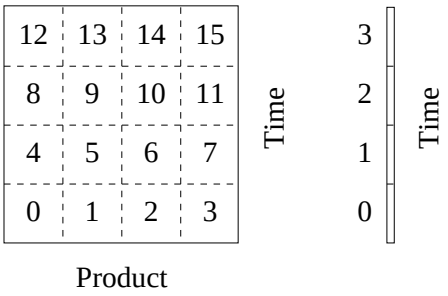

Fig. 1. Chunks at different levels.

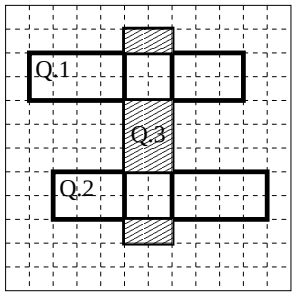

Fig. 2. Reusing cached chunks.

In the chunk-based caching scheme, query results to be stored in the cache are broken up into chunks and the chunks are cached. When a new query is issued, the query is analyzed to determine what chunks are needed to answer it. The cache is probed to find these chunks. Depending on what chunks are present in the cache, the list of chunks is partitioned into two. One part is answered from the cache. The other part consists of the missing chunks which have to be computed from the backend. To compute the missing chunks, a single SQL statement is issued to the backend translating the missing chunk numbers into the selection predicate of the SQL query.

Example 1. Figure 2 shows three queries Q1, Q2 and Q3 which are issued in that order. Q3 can use some of the cached chunks it has in common with Q1 and Q2. Only the missing chunks (marked by the shaded area) have to be computed from the backend.

An important property we use later is the closure property of chunks described in [DRSN98]. This means that there is a simple correspondence between chunks at different levels of aggregation. A set of chunks at a detailed level can be aggregated to get a chunk at higher level of aggregation. For example, Figure 1 shows that chunk 0 of $(Time)$ can be computed from chunks $(0, 1, 2, 3)$ of $(Product, Time)$.

3 Aggregations in the Cache

We now consider the problem of aggregation in more detail. In a multi-dimensional schema, there are many possible levels of aggregation, each of which corresponds to a different group-by operation. These group-bys can be arranged in the form of a lattice using the "can be computed by" relationship. This kind of structure has been extensively used in previous work [AAD+96][HRU96][SDN98].

For any group-by there are many group-bys from which it can be computed. In general, a group-by (x_1, y_1, z_1) can be computed from (x_2, y_2, z_2) if $x_1 \leq x_2$, $y_1 \leq y_2$ and $z_1 \leq z_2$. For example, group-by $(0, 2, 0)$ can be computed from $(0, 2, 1)$ or $(1, 2, 0)$. Thus we need to consider all the ancestors to determine if a particular group-by query can be answered from the cache.

Example 2. Consider a schema with three dimensions A, B and C. Dimension B has a two level hierarchy defined on it, whereas A and C have a single level hierarchy. Figure 3 shows the lattice formed by these dimensions. (x, y, z) denotes the level on each of the dimensions. $(1, 2, 1)$ is the most detailed level $(A_1B_2C_1)$ whereas $(0, 0, 0)$ is the most aggregated level $(A_0B_0C_0)$.

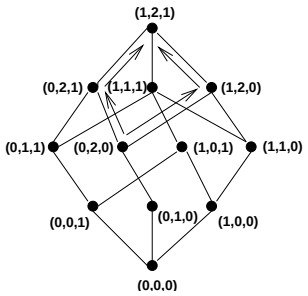

Fig. 3. Lattice of group-bys.

The problem becomes more complex when using the chunk-based caching scheme. Due to the closure property, there is a simple correspondence between chunks at different levels of aggregation. For example, a chunk at level $(0, 2, 0)$, say chunk 0, will map to a set of chunks at level $(1, 2, 0)$, say chunks 0 and 1. To compute chunk 0 of $(0, 2, 0)$ from $(1, 2, 0)$, we need both chunks 0 and 1 of $(1, 2, 0)$. It may happen that only chunk 0 of $(1, 2, 0)$ is present in the cache whereas chunk 1 is computable from other chunks. This implies that chunk 0 of $(0, 2, 0)$ is still computable from the cache. Thus to determine if a chunk of a particular group-by is computable from the cache, it is necessary to explore all paths in the lattice from that group-by to the base group-by. Figure 3 shows the different paths possible for computation of a chunk of $(0, 2, 0)$. Let us first examine a naive method of finding if a chunk is computable from the cache.

3.1 Exhaustive Search Method

The Exhaustive Search Method (ESM) is a naive implementation of finding if a chunk is computable from the cache. If a chunk is missing from the cache, it searches along all paths to the base group-by, to check if it can be computed from the cache. The algorithm is listed below:

```
Algorithm : ESM(Level, ChunkNumber)
Inputs:   Level – Indicates the group-by level
          ChunkNumber – Identifies chunk that needs to be computed
if (CacheLookup(Level, ChunkNumber)) // Lookup in the cache
    return true;
For each Parent Group-by in the lattice
    ParentLevel = level of the Parent Group-by
    ParentChunkNumbersList = GetParentChunkNumbers(ChunkNumber, Level, ParentLevel)
    success = true;
    For each chunk number CNum in ParentChunkNumbersList
        if (!ESM(ParentLevel, CNum))
            success = false
            break
    if (success)
        return true
return false
```

In the above algorithm, *GetParentChunkNumbers()* is a function which maps a chunk at one level to a set of chunks at a more detailed level. This algorithm searches different paths and quits as soon as it finds a successful path.

Lemma 1. *Consider a schema having n dimensions, with hierarchies of size h_i on dimension i. Let $(l_1, l_2, \ldots l_n)$ denote the level of a group-by. Note that $(0, 0, \ldots 0)$ is the most aggregated level and $(h_1, h_2, \ldots h_n)$ is the base level. The number of paths in the lattice for a group-by at level $(l_1, l_2, \ldots l_n)$ to the base level is given by:*

$$\frac{(\Sigma_{i=1}^{n}(h_i - l_i))!}{\Pi_{i=1}^{n}(h_i - l_i)!}$$

Proof. The proof follows from a simple combinatorial argument. We will skip it due to space constraints.

The actual number of recursive calls to ESM is much higher than this because a single aggregate chunk maps to multiple chunks at a detailed level (through the *MapChunkNumbers()* function) and ESM has to be called on each of those chunks, i.e. there is a fanout along each step of the path.

Lemma 1 suggests that the complexity of determining if a chunk can be computed from the cache depends on the level of aggregation of the chunk. For highly aggregated chunks, the number of paths searched is higher since there are many ways to compute them. For example, for the most aggregated level $(0, 0, \ldots 0)$, it is $(h_1 + h_2 \ldots + h_n)!/(h_1! * h_2! * \ldots h_n!)$. Note that this is the worst case complexity. The algorithm will complete as soon as it finds one way to compute the chunk. The average complexity depends on the actual contents of the cache.

4 Virtual Count Based Method

There is a lot of room for improvement in the naive ESM method. Our strategy, the *Virtual Count* based method (VCM) is motivated by two observations:

1. As ESM searches along different paths, a lot of vertices are visited multiple times due to the lattice structure.
2. A lot of the work can be reused by maintaining some summary of the state of the cache in terms of some meta-information about each chunk.

Example 3. Consider a small subsection of a lattice as shown in Figure 4. Suppose ESM is searching for chunk 0 at level $(0, 0)$. Two of the paths from $(0, 0)$ intersect at $(1, 1)$. As ESM searches these two paths, it will search for each of chunks 0, 1, 2 and 3 at level $(1, 1)$ two times – once for each path. It does not reuse the work done previously.

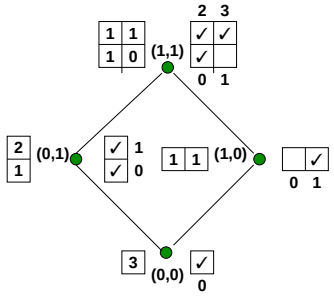

Fig. 4. Virtual counts.

In accordance with these observations, VCM maintains a count for each chunk at each group-by level. A chunk may be either directly present in the cache or may be computable through some path. Each path has to pass through some parent of that node in the lattice. Thus, the virtual count for a chunk is defined as:

Definition 1. Virtual Count: *The virtual count for a chunk indicates the number of parents of that node through which there is a successful computation path. The count is incremented by one if the chunk is directly present in the cache.*

The following property follows from the above definition:

Property 1. The virtual count of a chunk is non-zero if and only if it is computable from the cache.

Example 4. Figure 4 shows a very simple lattice with two dimensions having hierarchy size of 1 each. Level $(1,1)$ has 4 chunks, levels $(1,0)$ and $(0,1)$ have 2 chunks each and level $(0,0)$ has only 1 chunk. The figure shows the chunks that are present in the cache and the counts maintained by the VCM. Chunk 0 at level $(1,1)$ has count 1 since it is present in the cache and that is the only way to compute it. Chunk 1 at level $(1,1)$ is not computable, nor is it present in the cache, thus giving a count of 0. Chunk 0 at level $(1,0)$ has a count of 1 even though it is not present in the cache. This is because there is a successful computation path through 1 of its parents, i.e. level $(1,1)$. Chunk 0 at level (0,0) is present in the cache. Also, there are successful computation paths through two of its parents. Thus the count for chunk 0 at level (0,0) is 3.

Algorithm : VCM(Level, ChunkNumber)

```
Inputs:   Level – Indicates the group-by level
          ChunkNumber – Identifies chunk that needs to be computed
if (Count(Level, ChunkNumber) == 0) // Count is the array of counts
    return false;                                          ...          (I)
if (CacheLookup(Level, ChunkNumber))
    return true;
For each Parent Group-by in the lattice
    ParentLevel = level of the Parent Group-by
    ParentChunkNumbersList = GetParentChunkNumbers(ChunkNumber, Level, ParentLevel)
    success = true;
    For each chunk number CNum in ParentChunkNumbersList
        if (!VCM(ParentLevel, CNum))
            success = false
            break
    if (success)
        return true
assert(false) // control should never reach here
```

This algorithm looks similar to ESM in structure. However, the check for Count to be non-zero in statement (I) acts as a short circuit to reduce the complexity. If a chunk is not computable from the cache, VCM returns in constant time (just a single count lookup). If a chunk is indeed computable, VCM explores exactly one path (the one which is successful). Unsuccessful paths are rejected immediately without exploring completely. Compare this with the factorial number of paths for ESM.

4.1 Maintaining the Counts

Maintenance of the virtual counts makes lookups instantaneous. However, it adds an overhead when chunks are inserted or deleted from the cache since counts have to be updated at that time. The update algorithm while adding a chunk is listed below:

```
Algorithm : VCM_InsertUpdateCount(Level, ChunkNumber)
Inputs:   Level – Indicates the group-by level
          ChunkNumber – Identifies chunk whose count needs to be incremented
Count(Level, ChunkNumber) = Count(Level, ChunkNumber) + 1
if (Count(Level, ChunkNumber) > 1) // Chunk was previously computable
    return
For each Child Group-by in the lattice
    ChildLevel = level of the Child Group-by
    ChildChunkNumber = GetChildChunkNumber(ChunkNumber, Level, ChildLevel)
    ChunkNumbersList = GetParentChunkNumbers(ChildChunkNumber, ChildLevel, Level)
    flag = true;
    For each chunk number CNum in ChunkNumbersList
        if (Count(Level, CNum) == 0)
            flag = false
            break
    if (flag)
        VCM_InsertUpdateCount(ChildLevel, ChildChunkNumber)
```

We will not go into formal proof of the correctness of the above update algorithm in maintaining virtual counts. However, we do comment on its complexity.

Lemma 2. *Suppose we are inserting a new chunk in the cache at level* $(l_1, l_2, \ldots l_n)$. *The number of counts updated is bounded by* $n * \Pi_{i=1}^{n}(l_i + 1)$.

Proof. The proof can be found in [D99].

The trick of the VCM algorithm is to maintain just sufficient information to determine if a chunk is computable, keeping the update cost minimal at the same time. The exact complexity of a single insert depends on the cache contents. The amortized complexity over all the inserts is much lower than this worst case complexity. This is because, the updates are propagated only when a chunk becomes newly computable. A chunk can become newly computable only once. It could be more if there are deletes also since a chunk can keep switching between computable and non-computable state. However we don't expect this to happen very often for each chunk. Typically a chunk insert will cause update to propagate to only one level. This is similar to B-Tree splits, where most page splits do not propagate more than one level. The counts also have to be updated when a chunk is thrown out of the cache. The algorithm for that is similar to the *VCM_InsertUpdateCount()* method both in implementation and complexity, so we will omit the details in this paper.

It can be shown from Lemma 1 and 2 that the worst case complexity of the ESM find is much higher compared to the update complexity of the VCM update. Again, due to space constraints we will skip the proof.

5 Cost Based Strategies

The ESM and the VCM algorithms find just one path for the computation of a chunk. Assuming a linear cost of aggregation, the cost of computing a chunk is proportional to the number of tuples aggregated. This assumption has been used previously, for solving the precomputation problem [HRU96][SDN98]. There may be multiple successful paths through which a chunk could be computed. Each path will have different cost of computation depending on what chunks are being aggregated along that path. Both ESM and VCM can be extended to find the least cost path.

Example 5. Consider the simple lattice shown in Figure 5. There are two paths for computation of chunk 0 at level $(0,0)$. One way is to aggregate chunk 1 at level $(1,0)$ and chunks 0 and 2 at level $(1,1)$. Another way is to aggregate chunks 0 and 1 at level $(0,1)$. The costs for these two options are different since the number of tuples being aggregated is different. In general, it is better to compute from a more immediate ancestor in the lattice, since group-by sizes keep reducing as we move down the lattice.

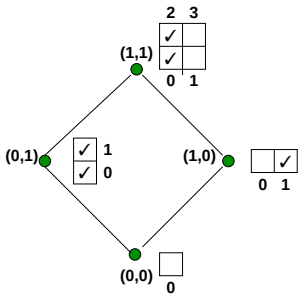

Fig. 5. Different costs of computation.

5.1 Cost Based ESM

The Cost based ESM (ESMC), instead of quitting after finding the first path, continues to search for more paths which might be of lesser cost. A listing of the ESMC algorithm is available in [D99]. The worst case complexity of ESMC is same as ESM. However, its average case complexity is much higher since it always explores all paths to find the minimum cost path. Whether this extra time is worth it depends on the time saved in aggregation. We present some experimental results which examine this in Section 7.

5.2 Cost Based VCM

The cost based VCM (VCMC) finds the best path for computing a chunk by maintaining cost information in addition to the count information. For each computable chunk, it stores the cost of the least cost path to compute it (`Cost` array) and the parent through which the least cost path passes (`BestParent` array). The find complexity is still constant time, which makes this method very attractive. The algorithm is similar to the VCM method and is listed in [D99]. The update algorithm is also similar to the one used by VCM. The only difference is that in VCMC an update is propagated in two cases – when a chunk becomes newly computable and when the least cost of computing a chunk changes. The worst case complexity of update remains the same, but the average complexity is slightly higher since an update is now propagated even when the least cost of a chunk changes.

Another advantage to maintaining the costs for the VCMC method is that it can return the least cost of computing a chunk instantaneously (without actually doing the aggregation). This is very useful for a cost-based optimizer, which can then decide whether to aggregate in the cache or go to the backend database.

6 Replacement Policies

The possibility of aggregating the cache contents to answer queries leads to interesting options for effectively using the cache space. In [DRSN98], we showed that a benefit based replacement policy works very well for chunks. In a simple cache, highly aggregated chunks have a greater benefit, since they are expensive to compute and thus are given a higher weight while caching. For aggregate

aware caching schemes, it is much more difficult to associate a benefit with a chunk. There are two reasons:

1. Other than being used to answer queries at the same level, a chunk can potentially be used to answer queries at a more aggregated level.
2. Whether a chunk can be used to answer a query at a more aggregated level depends on the presence of other chunks in the cache. This is because an aggregated chunk maps to a set of chunks at a more detailed level and all those chunks need to be present in order to compute the aggregated chunk. This also means that the benefit of a chunk is not constant but keeps changing as the cache contents change.

A detailed discussion of these issues can be found in [D99]. We will just summarize them here.

Example 6. In Figure 6(a), chunk 0 at level (1,1) has a lower benefit than chunk 0 at level (1,1) in Figure 6(b). The presence of chunk 1 at level (1,1) in Figure 6(b) leads to a higher benefit to both chunks 0 and 1, since they can now be used to compute chunk 0 at level (0,1).

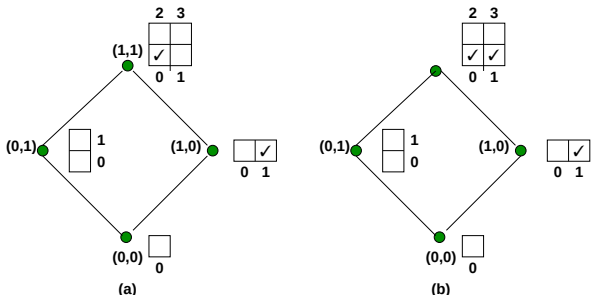

Fig. 6. Benefits.

6.1 Computing Benefits

The benefit of a newly computed chunk depends on how it has been computed.

1. *Cache computed chunk* – this is computed by aggregating other cached chunks. Its benefit is equal to the cost of this aggregation
2. *Backend chunk* – this is computed at the backend. Its benefit should also incorprate the cost of connecting to the backend, issuing a query and fetching results

Chunks which get computed at the backend should get a higher priority while caching than those which can be computed from other chunks already in the cache, since typically the overhead of fetching results from the backend is very high.

6.2 Forming Groups of Chunks

The optimal replacement policy should also try to form groups of useful chunks, since having a complete group leads to higher benefit for all chunks in the group as seen in Example 6. This is very difficult since it amounts to predicting what chunks are going to be inserted in the future. One way to solve this problem is to pre-compute entire group-bys and cache them. Since all the chunks in the group-by get cached, they can be used to compute any chunk at a higher aggregated level. In other words, pre-computing a group-by leads to the formation of useful group of chunks.

6.3 Two Level Policy

We propose the following policy in accordance with our observations:

- *Backend chunks* have higher priority and can replace *cache computed chunks*, but not the other way around. Replacement within each group is according to the normal benefit policy (i.e. highly aggregated chunks have higher benefit – same as that used in [DRSN98]).
- Whenever a group of chunks is used to compute another chunk, the clock value (we approximate LRU with CLOCK) of all the chunks in the group is incremented by an amount equal to the benefit of the aggregated chunk. This tries to maintain groups of useful aggregatable chunks in the cache.
- To help in the formation of useful groups, we pre-load the cache with a group-by that fits in the cache and has the maximum number of descendents in the lattice. Picking such a group-by will enable answering queries on any of its descendents.

Clearly this replacement policy is not optimal, and improving on it is fertile ground for future work. But as our experiments show, this policy provides substantial benefits over such policies as simple benefit based LRU.

7 Experiments

In this section, we describe the experiments used to evaluate the performance of the different schemes. We have a three tier system. The client issues queries to the middle tier which does caching of results. Both the client and the middle tier were running on a dual processor Pentium 133 MHz machine with 128 MB of main memory running SunOS 5.5.1. The backend database used was a commercial database system, running on a separate machine (a Sun UltraSparc 200 MHz with 256 MB of main memory running SunOS 5.5.1). A buffer pool size of 30 MB was used at the backend. The chunked file organization was achieved by building a clustered index on the chunk number for the fact file. The query execution times reported are the execution times at the middle tier.

All experiments were run on the APB-1 Schema [APB]. APB-1 is an analytical processing benchmark developed by the OLAP Council. APB has five dimensions with hierarchies – *Product*, *Customer*, *Time*, *Channel* and *Scenario*. Also, there is a measure *UnitSales* associated with the dimensions *Product*, *Customer*, *Time* and *Channel*. This mapping is stored in a fact table *HistSale*.

All queries are on the fact table *HistSale*, and ask for sum of *UnitSales* at different levels of aggregation. The number of nodes in the APB lattice is $(6 + 1) * (2 + 1) * (3 + 1) * (1 + 1) * (1 + 1) = 336$, since the hierarchy sizes are 6, 2, 3, 1 and 1 respectively. The data was generated using the APB data generator [APB], with the following parameters: number of channels = 10 and data density = 0.7. The table HistSale had about a million tuples, each of 20 bytes giving a base table size of about 22MB. The estimated size of the full cube for this schema is 902 MB. We experimented with cache sizes of 10 MB to 25 MB which is quite small compared to the size of the full cube. We performed two kinds of experiments. For one set (unit experiments) we used a very precise set of input queries, which were designed to bring out the best case, worst case and the average behavior. For the other set, we generated an artificial query stream. We describe the experiments in brief here. More details can be found in [D99].

7.1 Unit Experiments

The details of the first two experiments can be found in [D99]. We just summarize the results here.

Benefit of Aggregation This experiment demonstrates the benefit of implementing aggregations in the cache. We found that, on the average, aggregating in cache is about 8 times faster than computing at the backend. Note that this factor is highly dependent on the network, the backend database being used and the presence of indices and pre-computed aggregates.

Aggregation Cost Optimization This experiment measures how aggregation costs can vary along different paths in order to determine if cost based optimization is important. The difference between the fastest path and the slowest path is more for highly aggregated group-bys and lower for detailed group-bys. Experiments show that the average factor over all the group-bys was about 10.

Lookup Times In this experiment we measured the lookup times for all four algorithms ESM, ESMC, VCM and VCMC. We measured the lookup time for one chunk at each level of aggregation. The lookup time depends on the level of aggregation as well as on the cache contents. Table 1 lists the minimum, maximum and the average lookup times over all the group-bys for two cases: one where the experiment was run with an empty cache and the other where the cache was warmed up with all the base table chunks.

In both cases, the cache lookup times for VCM and VCMC are negligible. These methods explore a maximum of one path. The times for the ESM and ESMC are more interesting. When the cache is empty, none of the paths will be successful. However, both these methods have to explore all the paths. For detailed level group-bys the lookup time is low since very few paths of computations exist. For aggregated group-bys the time is much higher due to explosion in the number of paths. This variation is one of the drawbacks of the exhaustive methods since query response time will no longer be consistent. VCM and VCMC do not have these problems.

When all the base table chunks are in the cache, lookup times for ESM becomes negligible, since the very first path it explores becomes successful (base table makes all paths successful). For ESMC, the lookup time is unreasonable when all the base level chunks are cached, since ESMC has to explore all the paths, to find the best cost path. The cost of each path itself becomes much higher, since each chunk on each path is computable and the ESMC is called recursively on it. We have ignored this fanout factor (one chunk at a particular level maps to a set of chunks at a more detail level) in estimating the complexity of the lookup. Even a savings in aggregation costs cannot justify such high lookup times. So we do not consider the ESMC method in any further experiments.

	Cache Empty			Cache Preloaded		
	Min	Max	Average	Min	Max	Average
ESM	0	106826	1896.1	0	44	4.54
ESMC	0	134490	2390	0	19826592	272598
VCM	0	0	0	0	62	6.32
VCMC	0	0	0	0	149	13.15

Table 1. Lookup times (ms).

Update Times The VCM and VCMC method incur an update cost while inserting and deleting chunks, since they have to maintain count and cost information. ESM and ESMC do not have any update cost. Lemma 2 suggests

that update complexity is higher for more detail level chunks. To look at the worst case behavior we loaded all chunks of the base table – level (6,2,3,1,0) followed by all chunks at level (6,2,3,0,0). The update times vary while inserting different chunks, since how far an update propagates depends on what has been inserted in the cache previously. For example, while inserting the last chunk at level (6,2,3,1,0), update propagates all the way, since a lot of aggregate chunks become computable because of it. Table 2 shows the maximum, minimum and the average update time for the VCM and VCMC method. Even the maximum update time is quite feasible and on an average the cost is negligible. We can observe an interesting difference between VCM and VCMC. When inserting chunks at level (6,2,3,0,0), the update times for VCM are 0. All the chunks are already computable due to previous loading of level (6,3,3,1,0), so the updates do not propagate at all. However, for VCMC, insertion of chunks of (6,2,3,0,0) changes the cost of computation for all its descendents in the lattice. The cost information needs to be changed and the update costs reflects this.

	Loading (6,2,3,1,0)			Loading (6,2,3,0,0)		
	Min	Max	Average	Min	Max	Average
VCM	0	19	1.797	0	0	0
VCMC	1	36	5.427	0	15	10.09

Table 2. Update times (ms).

ESM	0
ESMC	0
VCM	32256*1 = 32 KB
VCMC	32256*6 = 194 KB

Table 3. Maximum Space Overhead.

Space Overhead The improved performance of VC based methods comes at the expense of additional memory required for the Count, Cost and BestParent arrays. The number of array entries is equal to total number of chunks at all possible levels. This might seem large, but it is feasible since the number of chunks is much smaller than the actual number of tuples. For example, in the schema used for our experiments, the base table had one million tuples of 20 bytes each. The total number of chunks over all the levels is 32256. Also, sparse array representation can be used to reduce storage for the arrays. Table 3 shows the maximum space overhead for the different methods assuming 4 bytes to store the cost and 1 byte each for the count and bestparent. Even for VCMC, the maximum overhead is quite small (about 0.97%) compared to the base table size. We can expect space overhead to scale linearly with the database size, since the number of chunks typically increases linearly assuming the average chunk size is maintained.

7.2 Query Stream Experiments

For these experiments, we generated a stream of queries using different parameters. These were similar to the ones used in [DRSN98]. For each experiment the cache was pre-loaded with a group-by as our "two-level policy". Performance is measured as an average over 100 queries. The cache sizes used ranged from 10 MB to 25 MB.

Generating a Query Stream The query stream is a mix of four kinds of queries – **Random**, **Drill down**, **Roll Up** and **Proximity** which try to model an OLAP workload. Roll-up, drill-down and proximity queries give rise to some locality in the query stream. While traditional caching can exploit proximity locality, we need active caches with aggregation to improve performance of roll-up queries. The query stream we used had a mix of 30% each of drill-down, roll-up and proximity queries. The rest of them (10%) were random queries.

Replacement Policies This experiment was designed to compare the "two level policy" described in Section 6 with the plain benefit based policy. Figure 7 plots the percentage of queries which are complete hits in the cache for different cache sizes. By "complete hits" we mean queries which are completely answered from the cache, either directly or by aggregating other chunks. The average query execution times are plotted in Figure 8. As the cache size increases, the percentage of complete hits increases. There are two reasons: the cache can be pre-loaded with a larger group-by (so more queries can be answered by aggregating) and just more chunks are cached leading to better hit ratio in general. The results show that the "two level policy" performs better. The main reason for this is that it has a better complete hit ratio. For example, consider the case when the cache is large enough (25 MB) to hold the entire base table. The "two level policy" caches the the entire base table, leading to 100% complete hit ratio. The ratio is lower for the other case (since it throws away useful base chunks in favor of computed chunks), causing it to go to the backend for some queries.

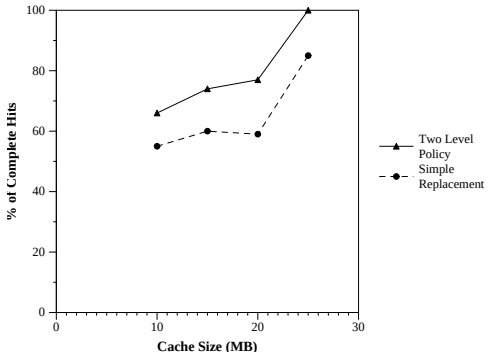

Fig. 7. Complete hit ratios.

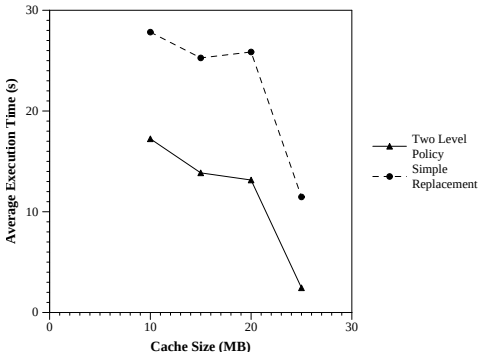

Fig. 8. Average execution times.

Comparison of Different Schemes In this experiment, we compared the different approaches to caching: no aggregation, ESM and VCM based methods. Experiments in Section 7.1 showed that the lookup times for ESMC are unreasonable. Also, the lookup and update costs of VCMC are comparable to that of VCM. So we consider only ESM and VCMC. The case with no aggregations in the cache is considered to demonstrate the benefit of having an active cache. Fig 9 shows the average execution times for running the query stream described earlier. The "two level policy" was used for replacement for the ESM and VCMC methods. However, for the no aggregation case, the simple benefit based policy was used since detail chunks don't have any higher benefit in the absence of aggregation. Both ESM and VCMC outperform the no aggregation case by a huge margin. This is expected, since, without aggregation, the number of cache misses are large. In fact only 31 out of the 100 queries are complete hits in the cache. For ESM and VCMC, the number of complete hits are much more.

VCMC outperforms ESM, with the difference being more for lower cache sizes. It might seem that the difference is not large. However, this is because of the fact that we are plotting the average execution times over all the queries. The queries which have to go the backend take disproportionately larger time and that affects the average. The difference between VCMC and ESM is more

pronounced for queries which are complete hits. When the query stream has lot of locality we can expect to get many complete hits. So speeding up complete hit queries is critical for increased system throughput. Table 4 shows the percentage of queries that are complete hits and the speedup factor for these queries.

To further analyze the speedup, Figure 10 shows the average execution times for queries that hit completely in the cache. We split the total cost for each query into three parts: cache lookup time, aggregation time and update time (to add the newly computed chunks). The bars on the left are for the ESM method and those on the right are for VCMC method. Even though it seems that the execution times are increasing for larger caches, note that the times cannot be compared across different cache sizes. This is because the set of queries which are complete hits is different for different cache sizes. At lower cache sizes, the speedup is more. This is because for smaller caches, the cache cannot hold a lot of chunks. So ESM has to spend a lot of time in finding a successful path of computation. This is reflected in very high lookup times. Also, there is a difference in the aggregation costs for ESM and VCMC, since VCMC considers costs to find the best path of computation. As the cache size is increased, the lookup time for ESM reduces, since there are more successful paths. In fact, for a cache size of 25 MB, the entire base table fits in memory and the first path it searches is a successful path. So the find time becomes negligible. The performance difference is now only due to the difference in the aggregation cost. We can also observe that the update times for VCMC method are very small. The update times, for a cache size of 25 MB, are slightly higher since it holds all the base level chunks. Whenever a new aggregated chunk is added or removed, it changes the costs of computation for its descendent chunks and these costs have to be updated. The find times for VCMC are negligible.

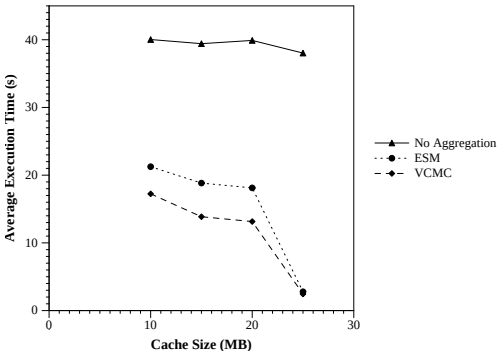

Fig. 9. Comparison of ESM and VCMC with No aggregation

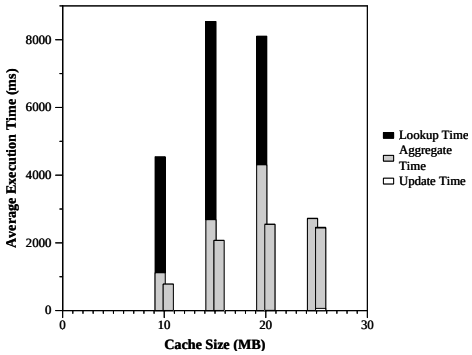

Fig. 10. Time breakup. Left side bars represent ESM and right side represent VCMC.

Cache Size (MB)	10	15	20	25
% of Complete Hits	66	74	77	100
Speedup factor (VCMC over ESM)	5.8	4.11	3.17	1.11

Table 4. Speedup of VCMC over ESM.

8 Conclusions

Providing a cache with the "active" capability of computing aggregates from cached data allows the cache to satisfy queries that would otherwise result in cache misses. Results from our implementation show that this can yield a substantial performance improvement over traditional caching strategies. The "two-level" policy works better than the simple benefit policy since it maintains useful groups of chunks and reduces accesses to backend. VCMC always performs better than ESM. When the cache size is small compared to the base (or "active" data size), the win of VCMC over ESM is more pronounced. A large part of this gain is due to savings in the find time and a smaller one due to aggregation time. When the cache is big enough to hold all base data and some more aggregated chunks, the gain in the find time is lost (since the first path chosen by ESM is successful). The improvement is now only due to the aggregation cost. So, in this case, we have a choice of using either ESM or VCMC depending on the locality in the query stream and the implementation effort one is willing to put in.

The area of active caching opens up a lot of opportunities for future work. One direction for such work would be to investigate the efficacy of such active caching approaches in workloads more general than those typically encountered in OLAP applications. There are also many interesting open issues for active caching for multidimensional workloads. One of the most interesting issues is that of cache replacement policies, since the problem is very complex for "active" caches.

References

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi. On the Computation of Multidimensional Aggregates, *Proc. of the 22nd Int. VLDB Conf.*, 506–521, 1996.

[APB] The Analytical Processing Benchmark available at *http://www.olapcouncil.org/research/bmarkly.htm*

[DFJST] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan. Semantic Data Caching and Replacement, *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[DRSN98] P. M. Deshpande, K. Ramasamy, A. Shukla, J. F. Naughton. Caching Multidimensional Queries Using Chunks, *Proc of ACM SIGMOD* , 259–270, 1998.

[D99] P. M. Deshpande. Efficient Database Support for OLAP Queries, *Doctoral Dissertation, University of Wisconsin, Madison.*, 1999.

[HRU96] V. Harinarayanan, A. Rajaraman, J.D. Ullman. Implementing Data Cubes Efficiently, *Proc. of ACM SIGMOD* , 205–227, 1996.

[KR99] Y. Kotidis, N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses *Proc. of ACM SIGMOD* , 371–382, 1999.

[RK96] R. Kimball. *The Data Warehouse Toolkit*, John Wiley & Sons, 1996.

[RSC98] K. A. Ross, D. Srivastava, D. Chatziantoniou. Complex Aggregation at Multiple Granularities, *Int. Conf. on Extending Database Technology*, 263-277, 1998.

[SDJL96] D. Srivastava, S. Dar, H. V. Jagadish and A. Y. Levy. Answering Queries with Aggregation Using Views, *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[SDN98] A. Shukla, P.M. Deshpande, J.F. Naughton. Materialized View Selection for Multidimensional Datasets, *Proc. of the 24th Int. VLDB Conf.*, 488–499, 1998.

[SLCJ98] J. R. Smith, C. Li, V. Castelli, A. Jhingran. Dynamic Assembly of Views in Data Cubes, *Proc. of the 17th Sym. on PODS*, 274–283, 1998.

[SSV] P. Scheuermann, J. Shim and R. Vingralek. WATCHMAN : A Data Warehouse Intelligent Cache Manager, *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[SS94] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays, *Proc. of the 11th Int. Conf. on Data Engg.*, 1994.