Parallel Querying of ROLAP Cubes in the Presence of Hierarchies

Frank Dehne
Carleton University
Ottawa, Canada
frank@dehne.net

Todd Eavis
Concordia University
Montreal Canada
eavis@cs.concordia.ca

Andrew Rau-Chaplin
Dalhousie University
Halifax, Canada
arc@cs.dal.ca

ABSTRACT

Online Analytical Processing is a powerful framework for the analysis of organizational data. OLAP is often supported by a logical structure known as a data cube, a multidimensional data model that offers an intuitive array-based perspective of the underlying data. Supporting efficient indexing facilities for multi-dimensional cube queries is an issue of some complexity. In practice, the difficulty of the indexing problem is exacerbated by the existence of attribute hierarchies that sub-divide attributes into aggregation layers of varying granularity. In this paper, we present a hierarchy and caching framework that supports the efficient and transparent manipulation of attribute hierarchies within a parallel ROLAP environment. Experimental results verify that, when compared to the non-hierarchical case, very little overhead is required to handle streams of arbitrary hierarchical queries.

Categories and Subject Descriptors

H.2.7.b [**Database Management**]: Data Warehouse and Repository; H.2.2.a [**Database Management**]: Access Methods

General Terms

Algorithms Design Performance

Keywords

Hierarchies, Caching, Data Cubes, Aggregation, Indexing, OLAP, Granularity, Materialization, Parallelization

1. INTRODUCTION

Online Analytical Processing (OLAP) has become an important component of contemporary Decision Support Systems (DSS). Central to OLAP is the data cube, a multidimensional data model that presents an intuitive cube-like interface to both end users and DSS developers. In recent years, the academic community has become increasingly interested in the cube model and a number of efficient cube generation algorithms have been presented in the literature[2, 16, 22].

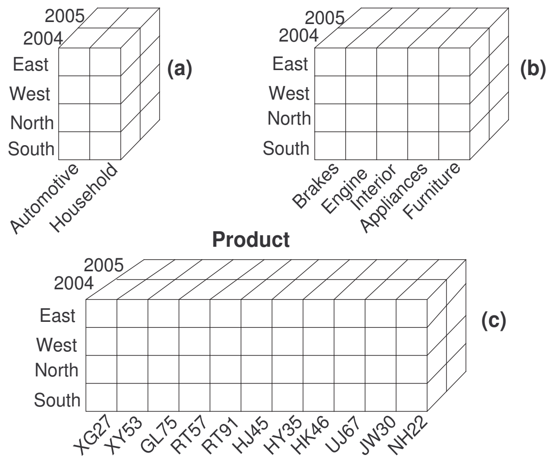


Figure 1: A hierarchical Product attribute broken down from (a) category, to (b) type, to (c) product number.

For the most part, the focus of these algorithms has been the generation of the cube data structure itself. Methods or techniques for efficient access/querying have received relatively little attention. When such methods have been presented, they typically assume the existence of non hierarchical attributes. In practice this is rarely the case. Figure 1 provides a simple example from the automotive industry. Here, we have three *feature* attributes — Product, Location, and Time — that can be viewed in terms of one or more *measure* attributes. In this case, each cell in the cube might be associated with an aggregated total for the measure attribute **Total Sales**. Note how the hierarchical Product dimension on the x-axis is broken down into increasingly finer levels of aggregation.

While it is possible to represent each of these hierarchical levels as a distinct feature attribute, doing so dramatically increases the complexity of the underlying problem. Specifically, the number of possible attributes or group-bys in a d-dimensional data cube is exponential on the number of dimensions. For example, a 10-dimensional cube would generate $2^d = 1024$ aggregated group-bys. By contrast, the total number of group-bys in the presence of hierarchies is given as $\prod_{i=1}^{d}(h_i + 1)$ when constructed from a data cube with d attributes, where dimension i has a hierarchy of size h [20]. The same 10-dimensional data cube with three-level hierarchies on each dimension would produce over one million group-bys. Clearly this is infeasible when the original input set may already contain terabytes of data.

An alternative approach to the generation and storage of fully materialized hierarchical cubes is to produce data cubes containing hierarchies represented only at the finest level of granularity. Hierarchical *roll up* or *drill down* is then done in real time during query resolution. In order for this to be feasible, the cube architecture must support both fast indexing and hierarchy-sensitive data structures. The associated overhead should be largely transparent to the end user.

In this paper we present a series of algorithms and data structures for the efficient manipulation of attribute hierarchies in “real time”. The framework has been integrated into the Sidera ROLAP Server, one component of the larger cgmCube Project [3, 5] that is designed to support terabyte scale data cubes. Our experimental results demonstrate that not only are the storage requirements — both in memory and on disk — quite modest but that real time processing overhead is likely to be imperceptible to the end user.

The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 3 we introduce the basic Sidera framework, while Section 4 describes the model and data structures for hierarchical attribute representation. Section 5 discusses the algorithms used by the Sidera parallel query engine. Section 6 presents our experiments and Section 7 the concluding remarks.

2. RELATED WORK

The data cube model was formally introduced in [9]. In the succeeding years, a series of algorithms for the efficient computation of the data cube were presented. Most were based in some way upon the cube *lattice* presented in [11] that identified the relationships between group-bys sharing common attributes. Various *top down* [1] and *bottom up* [2, 16] algorithms were developed, each exploiting some underlying sorting or hashing mechanism. Academic research has generally favored the relational or ROLAP approach, in which group-bys are stored in conventional table format. In [22], an array-based algorithm was presented. Though very efficient for dense, low dimensionality/cardinality data, this multi-dimensional or MOLAP model may be less scalable in large, sparse problem spaces.

Indexing the materialized cube has received less attention, despite its obvious effect upon performance. In [18], a number of conventional techniques including B-tree key concatenation and bit-mapped indexes are reviewed. Efficiency issues for high dimensional range queries are presented. True multi-dimensional index mechanisms offer greater potential. While dozens of such methods exist in the literature [7], the r-tree has arguably been the most promising [10]. OLAP-oriented r-tree usage is discussed in [17], with r-tree packing strategies presented in [12]. The related issue of caching for multi-dimensional OLAP queries was discussed in [6], but only in the context of MOLAP structures. The Dynamat System [13] provides a ROLAP-oriented model for dynamic view management and the caching of aggregation queries.

In terms of attribute hierarchies, published methods are even less common. Storage estimates for fully materialized hierarchies are presented in [20]. Perhaps the most interesting hierarchy-aware work is found in [21], where the authors propose a non-relational tree-based cube structure that eliminates prefix and suffix redundancies to create a cube data structure that is both compressed and searchable along attribute hierarchies. It is not clear, however, how amenable this structure is to complex range queries (as opposed to point queries) or the parallelization and external memory requirements of enterprise-scale data warehouses.

Parallelization for higher performance has also tended to focus upon cube generation [14, 15]. To our knowledge, the only true comprehensive parallel OLAP systems are described in [8, 4]. The first deals with the MOLAP framework while the second is ROLAP based.

3. PRELIMINARY MATERIAL

The data cube is a multi-dimensional model that supports an intuitive representation of core organizational data. For a d-dimensional space, $\{A_1, A_2, \ldots, A_d\}$ we have $O(2^d)$ attribute combinations or group-bys. Each of these k-attribute subsets, $k \subseteq d$, represents a unique aggregation of one or more feature attributes. We refer to the number of unique values in each of the d dimensions as the attribute cardinality $C_i, 1 \leq i \leq d$. The complete cube space is equivalent to the *cardinality product* $\prod_{i=0}^{d} C_i$. Large cardinality products are associated with *sparse* cube spaces.

While the cube can be described as a *logical* data model, it often forms a *physical* model as well, in that group-bys are often pre-computed so as to improve real time query performance. If the data is physically stored as a multi-dimensional array, we have a MOLAP design. MOLAP provides implicit indexing but performance sometimes deteriorates as the space — and the associated cube array — becomes more sparse (high dimensioanlity/high cardinality). Relational OLAP stores group-bys as distinct tables and scales well since only those records that actually exist are materialized and stored. However, it requires explicit multi-dimensional indexing in order to be used effectively.

3.1 Parallel ROLAP Architecture

Contemporary data warehouses have grown enormously in size, with the largest now pushing into the multi-terabyte range. For these massive data sets, multi-CPU systems offer great potential. The Sidera server was designed from the ground up as a high performance ROLAP indexing and query engine. The cube generation algorithms, which are part of the larger cgmCube system, are fully parallelized and are load balanced and communication efficient on both shared disk and shard nothing cluster architectures. Methods for both *full cube* (all 2^d views) and *partial cube* ($< 2^d$) materialization are supported.

Explicit multi-dimensional indexing is provided by a forest of parallelized r-trees. The r-tree indexes are *packed* using a Hilbert space filling curve so that arbitrary k-attribute range queries more closely map to the physical ordering of records on disk. The Hilbert-ordered records are striped across each of the p disks of the parallel machine where each striped partition forms a partial r-tree index.

Queries are distributed to each of the p nodes in parallel, allowing each of the processing nodes to participate equally in the resolution of every query. Load balancing errors due

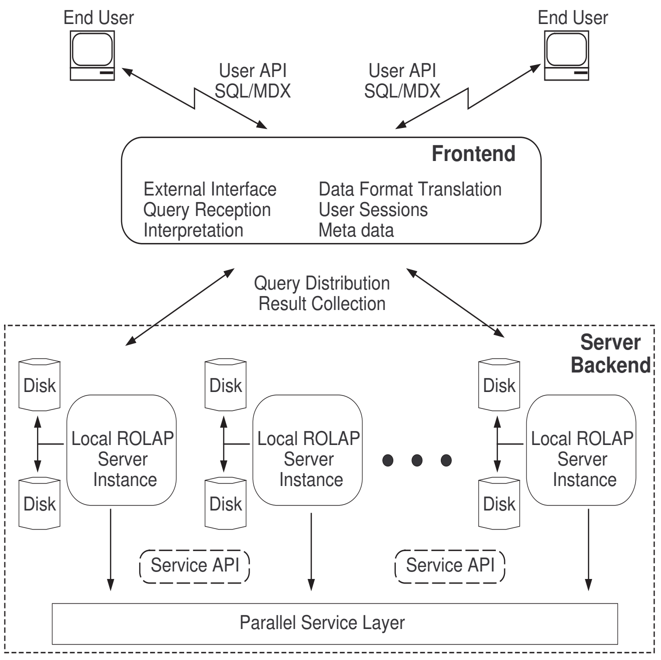


Figure 2: The Parallel Rolap Server architecture.

to set partitioning are typically less than 2%. In effect, each node serves as an independent ROLAP server, requiring no direct knowledge of peer processing nodes. Figure 2 provides a simple illustration of the hardware/software architecture for the Sidera query engine and the cgmCube system. Note that a Parallel Service API provides functionality (sorting, aggregation, communication, etc.) that allows local servers to operate independently.

3.2 Answering Queries

Before presenting the algorithms for hierarchical query resolution, we briefly discuss the core mechanisms for the original query engine. Algorithm 1 provides a high level description of Sidera's query resolution logic. The query interface is designed to be transparent, so that the user need not be aware of physical storage properties. We refer to this model as the *virtual data cube*. Queries are passed to each node so that a partial result can be computed. Because partial data cubes are often constructed in practice, the system may need to identify *surrogates* since the user-specified view may not physically exist. Differing sort orders may also need to be addressed. Queries are transformed appropriately and partial results are obtained. A highly optimized Parallel Sample Sort [19] forms the basis of the aggregation, merging, and ordering operations. Figure 3 provides a graphical illustration of the Sidera resolution model. Note how the end result exactly matches the user query, regardless of internal data characteristics.

4. ATTRIBUTE HIERARCHIES

In practice, a dimension will often contain a hierarchy that represents a set of unique aggregation granularities on a given attribute. A hierarchy is constructed on top of a *base* attribute $A_{(1)}$, which can be interpreted as the finest level of granularity on that dimension. With our earlier

Algorithm 1 Outline of Distributed *Partial* RCUBE Query Resolution

```
Require: A set $S'$ of indexed group-bys, striped evenly
    across $p$ processors $P_1, \dots P_p$, and a multi-dimensional
    query $Q$.
Ensure: Query result deposited on front-end or distributed
    across the $p$ processors.
 1: Pass query $Q$ to each of the $p$ processors.
 2: if the attributes in $Q$ match those of disk view $T$ then
 3:     select $T$ as the resolution target
 4: else
 5:     Locate surrogate group-bys $T$ containing a superset of
        the attributes in $Q$. Select the one with smallest size
        as the resolution target.
 6: end if
 7: Transform $Q$ into $Q\prime$ as per attribute ordering of $T$
 8: Add wildcard values for the peripheral attributes.
 9: In parallel, each processor $P_j$ retrieves records $R_j$
    matching $Q\prime$ for its local data and reorders the values
    of $R_j$ to match the order of $Q$. $P_j$ also removes the
    redundant values for the peripheral attributes of $T$.
10: Perform a Parallel Sample Sort of $R_1 \cup R_2 \cup \dots \cup R_p$
    with respect to the attribute ordering of $Q$. While per-
    forming the sort, aggregate duplicate records introduced
    by the peripheral attributes of $T$.
11: if the query result is to be deposited on the front-end
    then
12:     collect result via a MPI_AllGather ($p$ node transfer).
13: end if
```

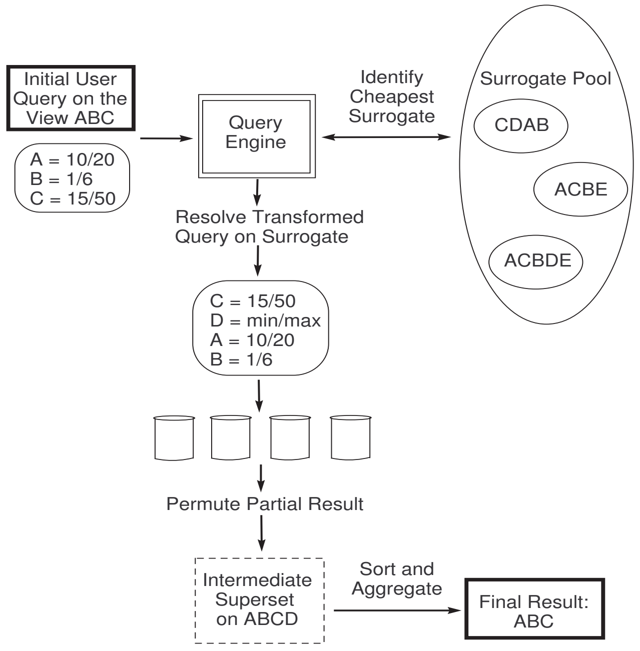


Figure 3: Basic query resolution, including surrogate exploitation.

example in Figure 1, the base attribute would be Product Number. The secondary attribute $A_{(2)}$ would be Product Type, while the tertiary attribute $A_{(3)}$ would be Product Category. Collectively, we refer to hierarchy levels above the base as *sub-attributes*. For a hierarchical attribute A, information captured by the attribute $A_{(i)}$ can always be obtained from $A_{(j)}$ when $i > j \geq 1$. This understanding is fundamental to the model presented in the remainder of this paper, in that data is stored only for the base attribute. As we will see, queries on other sub-attributes are *mapped* to this granularity level.

We now describe the notion of hierarchy *linearity*. First, note that $A_{(i)}$ is considered a *direct descendant* of $A_{(j)}$ if $A_{(i)}$ is the child of $A_{(j)}$ in the hierarchy. A hierarchy is linear if for all direct descendants $A_{(j)}$ of $A_{(i)}$ there are $|A_{(j)}| + 1$ values, $x_1 < x_2 \ldots < x_{|A_{(j)}|}$ in the range $1 \ldots |A_{(i)}|$ such that

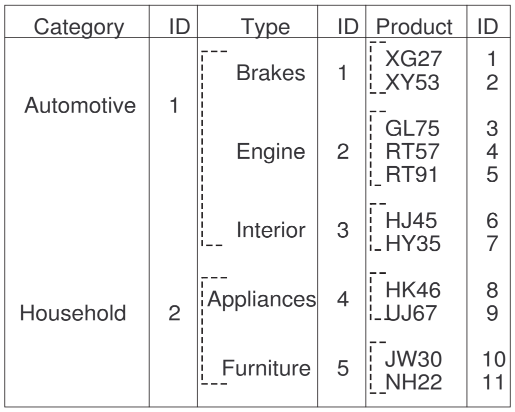
$$A_{(j)}[k] = \sum_{l=x_k}^{x_{k+1}} A_{(i)}[l]$$

where the array index notation [] indicates a specific value within a given hierarchy level. Informally, we can say that if a hierarchy is linear, there is a contiguous range of values $R_{(j)}$ on $A_{(j)}$ that may be aggregated into a contiguous range $R_{(i)}$ on $A_{(i)}$. As a concrete example, the Time hierarchy is linear in that a contiguous range of *day* values — say, 15 to 41 — can always be aggregated into a contiguous range of *month* values — in this case 1 to 2.

4.1 Preparing Hierarchies for High Performance ROLAP

The Time hierarchy is what we refer to as an *implicit* hierarchy, one whose linearity is self-evident. The linearity of other attributes is not always immediately obvious. With an alphanumeric Product Number, for example, it is not even clear how a Product Number such as "BY26T7999" compares to one like "GT45J7586" (in terms of $<$ *or* $>$ operations). The process of mapping ranges of Product Category or Product Type sub-attribute values to a corresponding range of Product Number values is therefore not clearly defined.

Note that we cannot simply make a linear pass through the native data set and assign identifiers to records simply based upon the order in which they appear. Hierarchical attributes mapped in this manner would be non-linear since an arbitrary mapping at the level of the base attribute would lead to non-contiguous ranges of non-base attributes. Instead, we enforce linearity by building mapping tables that are ordered by dimensions $A_{(k)} \times A_{(k-1)} \ldots A_{(1)}$. Figure 4 illustrates the mechanism for a three-level Product hierarchy — Product Number (base), Product Type (secondary), and Product Category (tertiary). The mapping table consists of a set of n records, with n equivalent to the cardinality C of the primary attribute $A_{(1)}$ (i.e., Product Number). That is, for each product number, we create a record containing the Product Number and the corresponding Type and Category. A k-dimensional sort — with primary attribute Category, secondary attribute Type, and tertiary attribute Number — is performed on the n records. Upon completion, we associate the distinct values of each column with consecutive integer IDs.



Category	ID	Type	ID	Product	ID
Automotive	1	Brakes	1	XG27	1
				XY53	2
		Engine	2	GL75	3
				RT57	4
				RT91	5
		Interior	3	HJ45	6
				HY35	7
Household	2	Appliances	4	HK46	8
				UJ67	9
		Furniture	5	JW30	10
				NH22	11

Figure 4: The mapping model, illustrated with a simple three level Product hierarchy.

4.2 hMap: A ROLAP Hierarchy Data Structure

The mapping mechanism creates a linear hierarchy on a multi-level OLAP dimension. In order to be used by the query engine, the model must be translated into an efficient in-memory data structure. In particular, the data structure must support the following *range translations*: (i) mapping from a base level attribute value $A_{i(1)}$ to the corresponding sub-attribute $A_{i(j)}, j > 1$; (ii) mapping from a sub-attribute $A_{i(j)}$ to the corresponding *range* on the base attribute $A_{i(1)}$.

The translation is accomplished with the multi-dimensional hMap data structure illustrated in Figure 5. Each core attribute A_i in the d-dimensional problem space is associated with $h_{A_i} - 1$ hierarchy maps, where h is the number of hierarchy levels for attribute A_i. No hierarchy map is associated with the base level of any hierarchy; these mappings are obtained indirectly. For a given level $A_{i(j)}, j > 1$, the associated map is made up of the maximum value from the range on $A_{i(1)}$ corresponding to the current value of $A_{i(j)}$. We use Figure 4 as an example. Type 2 (Engine) corresponds to the base level (Product ID) range $3 \mapsto 5$. The second cell of the Type map therefor contains the value 5.

Because of the significance of hierarchy mapping within the query resolution model, hMap access time is of primary importance. In this regard, we note that the worst case query time is bounded as $O(\log |l_{d(A_i)}|)$, where $|l_{d(A_i)}|$ is the cardinality of the destination level of the hierarchy on A_i. To see why this is the case, consider the following. To map from a sub-attribute $A_{i(j)}$ to the corresponding *range* on the base attribute $A_{i(1)}$, we simply index directly into the hMap using the value of $A_{i(j)}$ as the map index t. The contents of the associated cell represents the maximum range value, while $map[t-1]+1$ is the minimum value. This operation can be performed in $O(1)$ time.

By contrast, to map from a base level attribute value ε on $A_{i(1)}$ to the corresponding sub-attribute $A_{i(j)}, j > 1$, we must find the index position t, such that $map[t] >= \varepsilon$ AND $map[t-1] < \varepsilon$. Because the values of the map are sorted in ascending order, the query effectively reduces to a binary search on the destination map. The size of this map is $|l_{d(A_i)}|$. We therefore have a bound of $O(\log |l_{d(A_i)}|)$. Note as well that a mapping between arbitrary levels in the

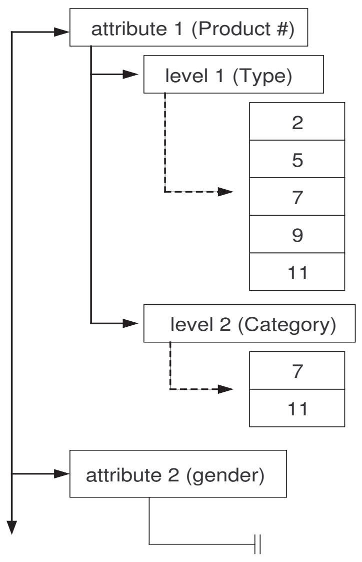


Figure 5: The hMap data structure, again using the Product hierarchy as an example.

hMap can be represented as an O(1) mapping to the base level, followed by a mapping to any non-base level.

A second consideration for the hMap is its memory requirements since we would like to save the bulk of these resources for buffering and query caching. Note that while we could guarantee O(1) for all operations on the hMap by including a base level map, the cardinality of the base level can be quite large. There might, for example, be a million or more Products. By eliminating the base, the collective size of a d-attribute hMap using non-base levels exclusively is just:

$$\sum_{i=1}^{d}\sum_{j=2}^{h_{A_i}} |l_{j(A_i)}|$$

where h_{A_i} is the number of levels in the hierarchy for attribute A_i and $|l_{j(A_i)}|$ is the cardinality of level j for the hierarchy on attribute A_i. In practice, this would likely be no more than a few dozen kilobytes for large data cube problems.

4.3 Caching Hierarchical ROLAP Queries

While parallel indexing facilities provide effective disk-to-memory transfer characteristics, optimal query response time relies to a great extent on an effective caching framework. Given the sizable memory capacity of the parallel ROLAP server, it is expected that a significant proportion of user queries will be answered in whole or in part from a *hot* cache.

Sidera provides a natively multi-dimensional, hierarchy-aware caching model. Specifically, resolved partial queries are cached on each node. For a new k-attribute range query, with ranges $R_1, R_2, \ldots\ldots R_k$, the cache mechanism must determine if, for each attribute A_i, the range R_i of the user query is a subset of the range on A_i of the cached query. If, for all k attributes, subset ranges are found, the cached query is used in place of a disk retrieval. At present, the Cache Manager does not process partial matches. That is, it does not answer queries partly from the cache and partly from disk. This, however, is the subject of ongoing research.

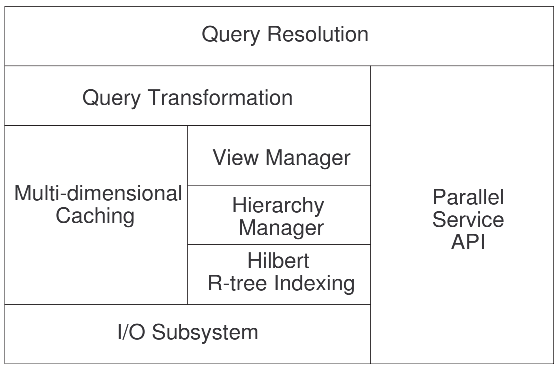


Figure 6: A block diagram of the module stack on each of the local processing nodes.

With respect to hierarchies, metadata is maintained by the Cache Manager and is used in conjunction with the hMap to perform translations between hierarchy levels. For a k-attribute user query, an arbitrary number of attributes can be re-mapped simultaneously. Note that queries are cached in their preliminary state — that is, they are cached in their base attribute form before final transformations have been applied. This permits hierarchies to be mapped to arbitrary levels — caching at levels above the base would prevent the cache from answering queries at finer levels of granularity.

It is important to note that the cache forms the basis of the core *Five Form* query model. Specifically, all OLAP servers should be able to support at least five basic OLAP-specific queries: roll-up, drill-down, slice, dice, and pivot. The query engine transparently manipulates the cache contents to further refine previous user queries. A drill down, for example, is produced merely by translating hierarchy levels within the current cache.

4.4 The Software Model

Taken collectively, the software architecture on each processing node forms a clean, modular design. Figure 6 illustrates how the hierarchy and caching components fit into the larger design. In the current context, the primary modules are the Hierarchy Manager, the Cache Manager, and the View Manager. The first two have already been discussed. The View Manager maintains meta data about the format and sort orders of views physically stored on disk. It is used when queries cannot be resolved from the Cache.

5. THE ALGORITHMS AND IMPLEMENTATION

The initial query engine, described in Section 3.2, was designed for simple, non-hierarchical attributes. With the addition of the hierarchy maps and caching framework, the algorithms had to be extended to accommodate more complex queries. Algorithm 2 describes the new algorithm for querying multi-dimensional data in the presence of hierarchies. Before processing, the query is transformed, taking into account hierarchical specifications. An initial result is obtained either from the cache or, if necessary, from disk. If obtained from the cache, the *prepareCachedQuery* function is used to re-order the cached attributes in the query buffer to match the order of the user query. Additional, non-specified attributes are dropped. If disk access is required, the initial data is retrieved via the r-tree indexes and is added to the cache. Query-specific post processing is then performed.

Algorithm 2 Query resolution in the presence of hierarchies

```
Require: A set $M$ of user-defined query parameters, a hierarchy manager hM, a cache Manager cM, and a view manager vM.
Ensure: Fully resolved and concatenated query result.
 1: load user query $uQ$ with parameter set $M$
 2: transformQuery(uQ, hM, vM)
 3: cached query $cQ$ = cM.checkForMatch ($uQ$)
 4: if $cQ$.cachedQuery != NULL then
 5:    temp buffer = prepareCachedQuery ($cQ$, $uQ$)
 6: else {otherwise, go to disk to answer the query}
 7:    initial results $I$ = processQuery ($uQ$)
 8:    add the initial results to the cache $cM$
 9: end if
    {do OLAP post processing}
10: result $R$ = postProcessing(uQ, hM, I)
11: if results required on front end then
12:    collect $R$ with MPI_Allgather
13: end if
```

5.1 Query Transformation

Algorithm 2 utilizes a function called **transformQuery** to convert the user query into a hierarchy-aware form that can be utilized by the query engine. This algorithm is described in Algorithm 3. The primary function is to create new range and hierarchy arrays. The range array provides the new high/low values for each of the A_i attributes in the user query. These are specified in terms of the base attribute. The hierarchy array will continue to reflect the hierarchy level requested by the user but will be updated with *wildcards* to indicate *full range* matching on peripheral attributes.

5.2 Post Processing

Once the initial result set has been constructed in Algorithm 2, post processing must be performed in order to produce the final result. This process is described in Algorithm 4. Note that the post processing routines are completely oblivious to the source of the initial result (cache or disk).

The **translateHierarchyValues()** function is used to map base level values in the initial result set into their appropriate counterpart at the destination level of the hierarchy (as defined by the user query). The system uses the Hierarchy Manager, hMap, and hierarchy array (constructed in Algorithm 3 for this purpose). A Parallel Sample Sort is performed to order records as per the user request and to permit efficient merging and aggregation. Note that the sorting subsystem is heavily optimized to minimize the movement of multi-value records. If surrogates or hierarchies have been specified, some form of additional aggregation will also be required. At this point, the result is ready for its return to the user.

Algorithm 3 Query Transformation Algorithm

```
Require: A user-defined query $uQ$ containing dimension set $M$, a hierarchy manager $hM$, and a view manager $vM$.
Ensure: Optimized query format.
 1: actual view $aView$ = qM.getDiskName($uQ$), where $aView$ contains dimension set $T$, $M \subseteq T$.
 2: create new attribute range array $newR$ of size $|T|$.
 3: create new hierarchy range array $newH$ of size $|T|$.
    {populate $newR$ and $newH$}
 4: for each attribute $i$ in $aView$ do
 5:    if $uQuery$ contains $aView[i]$ then
 6:       $low$ = range minimum for $aView[i]$ in $uQ$
 7:       $high$ = range maximum for $aView[i]$ in $uQ$
 8:       $l$ = hierarchy level for $aView[i]$ in $uQ$
 9:       if $l! =$ the base level then
10:          newR.low = hM.getBaseLow($aview[i]$, l, low)
11:          newR.high = hM.getBaseHigh($aView[i]$, l, high)
12:       end if
13:    else
14:       set high/low wildcards
15:    end if
16: end for
17: update the $uQ$ with $newR$, $newH$, and $aView$.
```

Algorithm 4 ROLAP Post Processing Algorithm

```
Require: query $uQ$, initial result $I$, hierarchy mgr $hM$
Ensure: final result $R$
 1: user-specfied view $uView = uQ.getUserView()$
 2: actual view $aView = uQ.getView()$
 3: if $uQuery$ contains hierarchies then
 4:    translateHierarchyValues(uQ, hM, I)
 5: end if
 6: do parallel sample sort
    {permute intermediate results as per user request}
 7: if surrogate used or hierarchy required (or both) then
 8:    $R$ = orderAndAggregate(I);
 9: else
10:    $R$ = arrangeSortedRecords(I);
11: end if
12: return $R$;
```

6. EXPERIMENTAL RESULTS

In this section, we provide experimental results that assess the ability of the query engine to efficiently support queries in the presence of hierarchies. We use synthetic data produced with our own data generator. Values are randomly generated and uniformly distributed. We note that while real data sets and/or skew patterns are important for other query evaluations, our objective here is to specifically assess the effect of hierarchy inclusion. As a result, synthetic data sets are sufficient.

We use a 10-dimensional fact table, with cardinalities arbitrarily chosen in the range 2–1000. The primary fact table used to compute the data cube consists of 1,000,000 records and, in turn, the materialized cube contains 1024 views and approximately 120 million records.

All tests are conducted with the Sidera engine running on a 16-node Linux cluster. Each node contains 1 GB of main memory and and a pair of 1.8 Ghz Intel processors. Disks are standard 40 GB drives and the nodes are connected by a 100 Mb Fast Ethernet switch.

6.1 Evaluation of Hierarchy Overhead

Hierarchies are managed in the system without any additional space or storage requirements. The only overhead is the run-time performance penalty associated with the mapping of queries to/from the base attribute. It is therefore important to evaluate the performance of queries associated exclusively with the base attribute versus those which are free to access arbitrary hierarchy levels.

Because individual millisecond-scale queries cannot be accurately timed, we use the standard approach of timing queries in batch mode. In our case, an automated query generator constructs batches of 1000 range queries, in which high/low ranges are randomly generated for each of k attributes, randomly selected from the d-dimensional space, $k \subseteq d$. Sort orders are also randomly determined. We note that this form of query generation actually overestimates query response time since users typically query low-dimensional views that can be easily visualized.

Figure 7 provides the test results. Here, we present the total response time for hierarchical versus non-hierarchical queries. (Results for 100,000 and 10 million records are also shown.) By non-hierarchical, we mean queries that have been restricted to the base attribute. The hierarchical queries have values selected from a randomly chosen hierarchy level. Hierarchies are defined on j attributes, $0 \leq j \leq k$. Five batches are generated and the average run-time is computed for each plotted point.

The graph demonstrates the modest degree of overhead that hierarchical transformation produces. In fact, at each of the three cube sizes, the total overhead averages less than 12%. Given that the parallel query engine processes approximately 100 queries per second for the 1 million record fact table, this added cost is likely to be negligible for the user.

6.2 Multi-dimensional Caching

In practice, OLAP queries tend to be iterative in nature. Users often define an initial exploratory query and then gradually refine the scope of the original query to obtain the desired result. Drill down, roll up, slice and dice, and pivot form the basis of such cube traversals. In the absence of a multidimensional, hierarchy-aware caching framework, the cost of Five Form processing is likely to grow significantly.

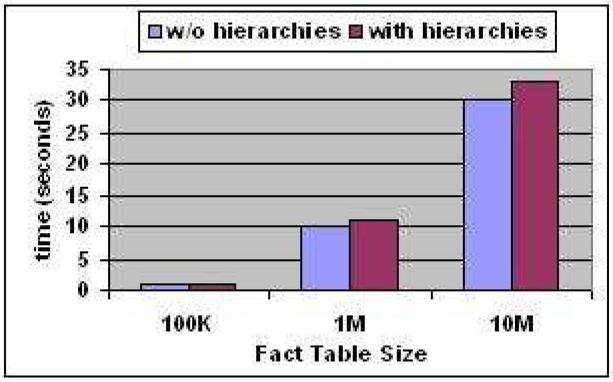


Figure 7: Comparison of hierarchical versus non-hierarchical queries for three cube sizes.

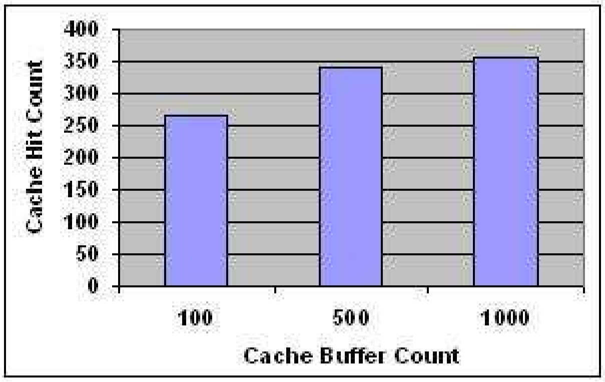


Figure 8: Comparison of cache hit rates for three buffer counts and batches of 1000 queries.

On average the caching design reduces query resolution time by 10-20% (for 1000 query batches) on the current system, depending upon the workload of the operating system and the sizes of its own disk caches. Iterative, hierarchy-based queries, in particular, are drawn exclusively from cache-managed memory. Even for *isolated* range queries, however, the cache framework is extremely effective. In Figure 8, we see the effect of caching on the cube generated from the 1M-record fact table. Again, batches of 1000 queries and the average of five runs are used. This time, however, the query generator randomly generates simple range queries only; iterative queries are not used. The objective is to determine the cache hit rate even if the canonical queries are absent.

For batches of 1000 queries, the graph shows the average hit rate as the number of available buffers increases from 100 to 1000. Specifically, the rate moves from 265 per 1000 queries to 355. Interestingly, a doubling of the buffer count from 500 to 1000 does relatively little as the 500-buffer model is able to achieve an average hit rate of 340. Again, we note that in practice the actual hit rate will be far higher than this since virtually 100% of the canonical OLAP query forms will be resolved directly from previously cached results.

7. CONCLUSIONS

The data cube has become an important theme in OLAP-based academic research. While a number of efficient algorithms for data cube generation have been presented in the literature, practical querying facilities have received less attention. Of particular importance is the ability to provide core OLAP query functionality on top of hierarchical feature attributes. In this paper, we present algorithms and data structures for hierarchical attributes that have been integrated into cgmCube's parallel data warehousing architecture. The methods do not require additional storage, instead relying on efficient mapping and transformation services that can be cost-effectively applied at run-time. In addition, a hierarchy-aware, multi-dimensional caching framework provides direct support for fundamental OLAP query types. Experiment results demonstrate the effectiveness of both mechanisms for arbitrarily generated query streams.

8. REFERENCES

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. *Proceedings of the 22nd International VLDB Conference*, pages 506–521, 1996.

[2] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proceedings of the 1999 ACM SIGMOD Conference*, pages 359–370, 1999.

[3] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Building large ROLAP data cubes in parallel. *International Database Engineering and Applications Symposium*, pages 367–377, 2004.

[4] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.

[5] F. Dehne, T. Eavis, and A. Rau-Chaplin. The cgmCUBE project: Optimizing parallel data cube generation for ROLAP. *Journal of Parallel and Distributed Databases*, 2005. To appear.

[6] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 259–270, 1998.

[7] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[8] S. Goil and A. Choudhary. High performance multidimensional analysis of large datasets. *Proceedings of the First ACM International Workshop on Data Warehousing and OLAP*, pages 34–39, 1998.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proceeding of the 12th International Conference On Data Engineering*, pages 152–159, 1996.

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, 1984.

[11] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, pages 205–216, 1996.

[12] I. Kamel and C. Faloutsos. On packing r-trees. *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[13] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *ACM Transactions on Database Systems*, (4), 2001.

[14] S. Muto and M. Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. *ACM 2nd Annual Workshop on Data Warehousing and OLAP*, pages 67–72, 1999.

[15] R. Ng, A. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. *Proceedings of 2001 ACM SIGMOD Conference on Management of Data*, pages 25–36, 2001.

[16] K. Ross and D. Srivastava. Fast computation of sparse data cubes. *Proceedings of the 23rd VLDB Conference*, pages 116–125, 1997.

[17] N. Roussopoulos, Y. Kotidis, and M. Roussopolis. Cubetree: Organization of the bulk incremental updates on the data cube. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 89–99, 1997.

[18] S. Sarawagi. Indexing OLAP data. *Data Engineering Bulletin*, 20(1):36–43, 1997.

[19] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1990.

[20] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *Proceedings of the 22nd VLDB Conference*, pages 522–531, 1996.

[21] Y. Sismanis, A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical dwarfs for the rollup cube. *DOLAP 03: Proceedings of the 6th ACM international workshop on Data warehousing and OLAP*, pages 17–24, 2003.

[22] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 159–170, 1997.