Serving Datacube Tuples from Main Memory

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

Kazi A. Zaman
Columbia University
zkazi@cs.columbia.edu

Abstract

Existing datacube precomputation schemes materialize selected datacube tuples on disk, choosing the most beneficial cuboids (i.e., combinations of dimensions) to materialize given a space limit. However, in the context of a data-warehouse receiving frequent "append" updates to the database, the cost of keeping these disk-resident cuboids up-to-date can be high. In this paper, we propose a main memory based framework which provides rapid response to queries and requires considerably less maintenance cost than a disk based scheme in an append-only environment. For a given datacube query, we first look among a set of previously materialized tuples for a direct answer. If not found, we use a hash based scheme reminiscent of partial match retrieval to rapidly compute the answer to the query from the finest-level data stored in a special in-memory data structure. Our approach is limited to the important class of applications in which the finest granularity tuples of the datacube fit in main memory. We present analytical and experimental results demonstrating the benefits of our approach.

1 Introduction

Datacube queries compute aggregates over database relations at a variety of granularities, and they constitute an important class of decision support queries. The databases may represent business data (such as sales data), medical data (such as patient treatments) or scientific data (such as large sets of experimental measurements).

An example of a typical datacube is: Broken down by latitude, longitude, solar-altitude and day, find the maximum cloud coverage, including subtotals across each dimension. This implies that in addition to computing the maximum cloud coverage over all the data, we would have also have to compute the maximum cloud-coverage over all latitudes, all longitudes, all latitude and longitude pairs etc. Since we have 4 dimensions there are 2^4 granularities at which we would have to compute subtotals. For ease of representation, the granularities that are aggregated over are replaced by ALL's in the result tuple. If we are computing the maximum cloud coverage over each latitude irrespective of longitude, solar-altitude or day, we are computing tuples of the form (`latitude`,ALL,ALL,ALL,t) where t is the computed aggregate.

Most OLAP systems precompute some or all of these aggregates to answer queries as quickly as possible. To answer all possible queries over the datacube, we could materialize the entire datacube and store it on disk. The tradeoff is that the datacube may be substantially larger than the base data and may require more space than available. This is especially true for large sparse datasets. In [7], a greedy algorithm is proposed which attempts to maximize the benefit of the set of aggregates picked. In a subsequent paper [5] techniques for the selection of indices in conjunction with the aggregates were presented. In [14] algorithms with faster running times but which achieve the same performance as [7] are developed.

These previously mentioned schemes materialize datacube tuples on disk and do not exploit the available main memory. Rapidly decreasing main memory prices have lead to workstations with over a gigabyte of RAM. The Asimolar Report on database research [3] states:

Within ten years, it will be common to have a terabyte of main memory serving as a buffer pool for a hundred-terabyte database. All but the largest database tables will be resident in main memory.

Under the circumstances today, even if we can fit the base table in main memory, we probably will not be able to fit all the tuples of the datacube. In this paper we develop a framework which enables us to efficiently answer queries under these circumstances by materializing a subset of datacube tuples and storing them in memory. An important benefit of utilizing main memory is that we can provide answers to queries rapidly without having to go to disk. Users are likely to require answers to their queries

*This research was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by an NSF Young Investigator Award, by NSF grant number IIS-98-12014, and by NSF CISE award CDA-9625374.

which are accurate up-to-the-minute. An advantage of our approach is that we do not have to update a large number of previously materialized tuples on disk whenever new tuples are available. We can, instead, efficiently update our in-memory data structures and provide the user with useful running aggregates.

The previous work cited has focussed on selecting datacube tuples for materialization at the cuboid level (we refer to the set of aggregates at a particular granularity as a cuboid). This means that either all the tuples of a cuboid are selected for materialization or none at all. In our framework we can materialize any number of tuples from any cuboid - our unit of materialization is a tuple. This enables us to efficiently answer queries which require a *slice* of a cuboid.

We use a two-level materialization scheme. Our level-1 store contains datacube tuples. Since we cannot expect to materialize all datacube tuples, we store only "high value" tuples in the level-1 store. We analyze what constitutes a "high value" tuple, and demonstrate that tuples with many ALLs and tuples with high query probabilities are good candidates for materialization.

Our level-2 store contains all tuples at the finest granularity. We store the data in a structure that allows a form of partial match query to answer queries without scanning the entire finest granularity dataset. We interrogate the level-2 store only if we find no match in the level-1 store. Our data structures enable fast incremental updates in response to new data, thus allowing the datacube server to supply up-to-date results.

We show how to optimize space between the level-1 and level-2 stores, and how to prioritize tuples for materialization in the level-1 store. Our experimental results show that if the available amount of memory is very small, priority is given to the level-2 store. Once we have an adequate amount of memory the level-2 store tends towards its natural size. The experimental results indicate that for skewed data the choice of a tuple as the unit of materialization is a good one. Our experimental results show that additional memory of a few megabytes enables us to reduce the response time to 2-4 ms for a typical example, which is considerably less than the time required to compute the tuple from the finest level data (194 ms) or to recover a materialized tuple from disk.

We believe that serving datacube tuples from RAM is feasible now for a variety of applications. Further, as main memories increase in size, our techniques will be applicable for more and more applications.

2 Notation, Terminology and Cost Models

The computation of a datacube query with d `CUBE BY` attributes $(B_1, B_2 \ldots B_d)$ involves computing the aggregates over a relation at 2^d granularities where each granularity is one of the the possible 2^d subsets of our d `CUBE BY` attributes. Attributes that are not present in such a subset are replaced by a special value ALL in the datacube result. We refer to each of these granularities as a *cuboid* and we use the notation $Q(\vec{B_i})$ to denote the cuboid at granularity $\vec{B_i}$. We assume that the aggregate function(s) to be computed are *distributive* or *algebraic* [4]. (For simplicity of presentation, we will describe the computation for a single distributive aggregate function; the extension to multiple aggregate functions is straightforward. In practice, one would probably compute several aggregates, such as sum, count, min, max and sum-of-squares, and derive other aggregates from these.)

2.1 Queries

For now, a "datacube query" (or just "query") is a request for a single tuple that may be in the datacube. The user specifies values for some of the attributes, and ALLs for the remaining attributes. The answer to the query is the datacube tuple with that combination of attribute values/ALLs.[1] We will consider more general notions of query later in Section 4.1.

There are a number of assumptions we could make about the distribution of queries over the datacube. Queries could select all the tuples from a particular cuboid or alternatively we could have *slice queries* [5] that select a particular tuple from a cuboid. Let the ith each cuboid in the datacube have probability p_i of being queried where for a d attribute dataset $\sum_{i=1}^{2^d} p_i = 1$.

We take the most general model of query probability distribution, namely that the jth tuple in cuboid i has a probability $t_{i,j}$ of being queried, where $\sum_j t_{i,j} = p_i$. Our only restriction is that these $t_{i,j}$ probabilities can be calculated either during or immediately after the datacube computation itself, before any queries are actually posed.

Several kinds of query distribution can be justified as reasonable. Both [7, 14] make the assumption that each cuboid has the same probability of being queried, i.e., the p_i values are equal. For each i, the $t_{i,j}$ values would also be equal. We call this probability model the *uniform cuboid* distribution.

A second kind of query distribution would assume query probabilities are proportional to a tuple's *count*. We define the *count* of a datacube tuple to be the number of tuples from the base relation R which would need to be aggregated to compute the aggregate associated with the tuple. For example, the count of the tuple $< ALL, ALL, \ldots, ALL >$

[1] If the particular combination of attributes in the query does not correspond to an actual datacube tuple, then we can either return a special flag indicating "no tuple," or return a tuple with default aggregate values (e.g., 0 for a count aggregate).

would be $|R|$, the number of tuples in the base relation R. (For any cuboid in the datacube, the sum of counts of tuples in it will be equal to $|R|$.) We call this the *count based* distribution.

The motivation for a count based distribution is that users might be most interested in values of tuples for which the most data is available. If an example attribute *state* takes the value of one of the 50 states of the USA, queries will be more likely to specify those states which have substantial amounts of data rather than ones for which little information is available.

A third kind of distribution would be to specify a precise weighted query workload. Based on the workload, appropriate probabilities can be assigned to each tuple in the datacube at materialization time. We call this a *workload-based* distribution. Note that, unlike the other distributions considered so far, a workload-based probability distribution might give nonzero query probability to tuples that are not in the datacube. For example, the workload might contain a query like "Give the total sales in Idaho of green BMW cars, irrespective of month sold," even when there are no actual sales of green BMWs in Idaho. Further, as we shall see, it may still be beneficial to explicitly store a tuple indicating this fact in the level-1 store, rather than requiring a search of the level-2 store to answer this query.

A special kind of workload-based distribution is one composed of full granularities. A *full granularity* is a collection of equiprobable queries at a granularity of the datacube where all the non-ALL attributes range over all values from the appropriate domain. We call this a *full-granularity* distribution. This distribution is different from a uniform cuboid distribution that assigns equal probabilities to datacube tuples at a granularity; there may be fewer tuples in the cuboid than queries in the corresponding full granularity.

For example, a full granularity such as "For each car in domain C of cars, and for each state in domain S of states, give the total sales of that car in that state" defines a workload of $|C| \cdot |S|$ equiprobable queries. If a particular car was not sold in some state, then the uniform cuboid distribution would have fewer than $|C| \cdot |S|$ queries. Full granularity workloads are likely to be common, because they correspond to filling multidimensional grids that are frequently used in data analysis.

Each of these three distributions satisfies the condition that "probabilities can be calculated either during or immediately after the datacube computation itself, before any queries are actually posed." In the case of the count-based distribution, we can compute the count aggregate within the datacube computation. On the other hand, a query distribution that becomes apparent only over time as queries are posed does not satisfy our condition. We need to know tuple probabilities in order to properly configure our system.[2]

In what follows, we shall abuse terminology slightly by talking about query tuples "in the datacube" or "in a particular cuboid." For workload-based distributions, we must interpret these statements as including queries at the appropriate granularity that may not actually be in the datacube itself.

2.2 Cost Model

In [7], the cost model is a linear cost model where the cost of answering a query Q is the number of rows which need to be read in order to answer the query. This cost model is simplistic and does not take into account whether we have materialized the aggregate in memory or on disk. If the aggregate is to be computed from the base relation on disk, the number of disk I/O's will depend upon which attributes the relation has been clustered. In [5] the same cost model is used for indices too. A more appropriate cost model is to express the cost of answering a query in terms of the number of memory accesses and the number of disk I/O's required. Since our framework does not involve accessing disk, our analysis will be in terms of number of memory accesses required.

Another assumption made in earlier work is that the cost of answering a query on a cuboid which has not been materialized is equal to the cost of scanning the base table, i.e the number of rows in the table. This assumption will not be true if (as we shall do in this paper) we organize the base data in such a way that we can treat each query for a non materialized tuple as a partial match retrieval query against the base data and a subsequent aggregation of the result tuples.

3 The Tuple Serving Framework

3.1 Problem Description

Our setting of the problem is as follows. Given main memory M, and a base relation R, how can we appropriately utilize memory to minimize the average cost of executing a query over the d dimensional datacube D constructed over R according to a defined cost model while simultaneously providing update performance at worst $O(log|R|)$. Our problem addresses the selection of tuples to materialize, the definition of appropriate data structures, and the optimization of the system's configuration.

The problem is relatively simple for smaller values of d with moderate attribute cardinalities. In this case we can store a d dimensional array in memory where each

[2] If the query distribution is unknown, then our framework could be used with the level-1 store being a cache of previously asked queries. Dynamic configuration of such a system is beyond the scope of this paper.

dimension takes each possible value (including ALL) of the corresponding attribute. This solution does not scale with larger values of d and high attribute cardinalities since the size of the array will become prohibitively large.

3.2 Overall Approach

The overall approach is to exploit the following key points which hold for sparse skewed data.

- We store the finest granularity cuboid, since it cannot be computed from other cuboids. One should use a data structure that allows the computation of coarser granularity cuboids without a complete scan of the finest-granularity data.
- The coarsest granularity cuboids (i.e., those with many ALLs) are the most expensive to compute from the base data. On the other hand, the space needed to explicitly store these datacube tuples is relatively small.
- Skewed datasets and/or query probability distributions produce cuboids where the tuple probabilities are skewed. In such situations it is beneficial to selectively materialize high probability tuples.

We have a two tier framework (Figure 1) for handling this problem. We consider a tuple to be a $< K, V >$ pair where K is a composite key constructed from the d attribute values and V is the value to be aggregated. The tuples in the level-1 store may be from any cuboid in the datacube. The level-1 store is represented as a hash table. Given a query q, we first check if the result tuple has been materialized in the level-1 store by hashing on the composite key specified in q. If not, we can compute it from the finest cuboid tuples in the level-2 store.

Our level-2 store shown in Figure 1 is similar to that used for partial match retrieval. We have an array of n slots in memory. A set of functions $h_i, i = 1, \ldots, d$ together map each possible tuple to one of these slots. Each finest level tuple is represented as an element in a linked list whose head pointer is in the slot array.

Note that tuples in the level-1 store can have ALLs as attribute values, while tuples in the level-2 store cannot.

When a query is answered using the level-2 store, we want to examine just a fraction of the slots. If the query has all its attributes specified we would like to be able to look in exactly one slot. If some of the attributes in the query are not specified, we would like to use the known attribute values to limit the set of slots we need to search. Note that the number of slots is likely to be many orders of magnitude smaller than the product of the cardinalities of the attribute domains.

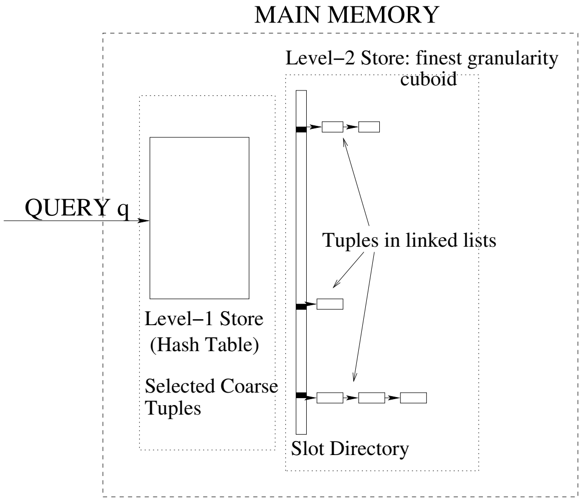

Figure 1. Framework for answering queries

Suppose that we have space for a slot array with up to T entries. (We'll see how to choose T later.) Then the 1-dimensional slot array can be visualized as a d-dimensional array `S` with array bounds of $b_1, \ldots, b_d$ in each of the d dimensions, as long as the product $b_1 b_2 \cdots b_d \leq T$.

We construct d domain mappings $h_1, \ldots, h_d$, where h_i maps values in the domain of attribute i into the range $1, \ldots, b_i$. A tuple $(v_1, \ldots, v_d, x)$ with attribute values v_i and aggregate value x maps to the slot `S[`$h_1(v_1)$`][`$h_2(v_2)$`]...[`$h_d(v_d)$`]`. This tuple is placed in the list whose head pointer is stored in that entry of the slot array. All finest-granularity tuples are placed in the appropriate list.

Consider a query that specifies all but two of the attributes (suppose the first two attributes are ALLs in the query). To answer this query we need to examine only $b_1 b_2$ lists. On average, that means we can ignore $1/(b_3 \cdots b_d)$ of the finest-level data, a significant improvement over a full scan.

There are a number of design issues to be addressed before this framework can be implemented. We haven't yet specified how the b_i values are chosen, nor how the h_i functions are constructed. We also haven't yet shown how to derive T, the preferred size of the slot array, nor how to choose the tuples for the level-1 store. Each of these issues will be addressed in the following sections.

3.3 Optimizing the Level-1 Store

If there is sufficient space, one could conceivably store all of the datacube in the level-1 store. Given sufficient memory, we would do just that. However, we expect that we won't have sufficient memory to store the complete datacube since the datacube is orders of magnitude bigger than the original dataset given sufficiently large attribute cardinalities and a sufficiently large number of dimensions.

Our intuition suggests two guidelines for choosing tuples for the level-1 store. The first guideline says "Materialize tuples with the most ALLs first." The reasoning behind this guideline is that (a) the size of these cuboids will be smaller than the size of cuboids with fewer ALLs, so we get better space utilization, and (b) queries with more ALLs are more expensive in the level-2 store: if we can satisfy these queries in the level-1 store, then we won't need to interrogate the level-2 store.

The second guideline says "Materialize tuples with the highest probabilities first." The main reason for this guideline is that those tuples are more likely to be queried according to our cost model.

The two guidelines are often compatible, since datacube tuples with more ALLs will (in several query distribution models) tend to have higher probabilities. However, we do need to prioritize them in order to determine whether or not to materialize a high-probability tuple with fewer ALLs in favor of a low-probability tuple with more ALLs. We shall address this issue in Section 3.5.

We notice that if a tuple is in the level-1 store, it is very likely that all coarser tuples are in the level-1 store. Thus, it doesn't pay to try to aggregate tuples in the level-1 store in response to a query. This is borne out by experiments described in Section 5.

3.4 Optimizing the Level-2 Store

Our level-2 store has T slots. Given a partial match query where some attributes are unspecified, we can efficiently reference the slot addresses which have to be checked by using an appropriate number of nested for-loops. Within the for loops, we traverse the list and accumulate all tuples that match the query. Note that the h_i functions map attributes to a smaller domain, so there may be nonmatching tuples in the list.

Example 3.1: Consider a query on an 8-dimensional dataset in which the first two dimensions are ALLs, and the other six are specified. Using the terminology of Section 3.2, suppose the specified attributes $v_3, \ldots, v_8$ are mapped to `y3`, …, `y8` via $h_3, \ldots, h_8$ respectively. Let `b1` and `b2` be b_1 and b_2. Then the query can be answered according to the pseudo-code in Figure 2 that assumes arrays start at index 1. We have a total of $b_1 b_2$ lists to traverse. □

Choosing the b_i Values

We now address the question of how to choose the b_i values. At one extreme, we could give high values to some attributes and low values to others. At the other extreme, we could try to balance all b_i values among the various attributes. Which is more appropriate?

An answer to this question appears to require a weighted average of the cost of all queries that reach the level-2 store. However, this weighted sum has a special form that makes it amenable to analysis. All terms in the sum include a product of some of the b_is. Consider b_1 and b_2. The terms fall into four categories: those that include both b_1 and b_2, those that include neither, those that include just b_1, and those that include just b_2. In the case that queries on all cuboids are equiprobable (e.g., the uniform cuboid distribution or the count based distribution) there is symmetry among all the b_is. By symmetry, there is a one-to-one correspondence between terms including just b_1 and terms including just b_2. Thus the overall weighted sum has the form

$$b_1 b_2 X_1 + (b_1 + b_2) X_2 + X_3$$

for some expressions X_1, X_2 and X_3. If the product $b_1 b_2 \cdots b_d$ is fixed, but b_1 and b_2 are allowed to vary subject to that constraint, which configuration minimizes the expression above? The only term that actually changes is the middle term $(b_1 + b_2)X_2$. On the real numbers, this term is minimized (subject to $b_1 b_2$ remaining fixed) when $b_1 = b_2$. If $b_1 \neq b_2$ then we can reduce the term by bringing b_1 and b_2 closer together. Since the choice of b_1 and b_2 was arbitrary, we can apply the same reasoning to all of the b_i values.

Thus we should aim to distribute the b_i values in a balanced fashion, with $b_i \approx T^{1/d}$. If an attribute cardinality is actually less than this computed b_i, then we can use the attribute cardinality and adjust the other b_j values upwards.

The intuition behind the choice of balanced b_i values is the following: Suppose we were to start with a balanced set of b_is, and then doubled b_1 while halving b_2. Then queries depending on b_2 would take twice as long, while queries depending on b_1 would take half as long. Adding up the cost of queries that depend on exactly one of b_1 and b_2 we get a cost equal to $(2 + 1/2)/2 = 1.25$ times the original cost, while the other queries do not change in cost.

In the general case, where queries on all cuboids are not equiprobable, the optimal set of b_is is obtained by solving a nonlinear integer programming problem. Because of the intractability of finding an optimal solution, heuristic approximations will usually have to suffice.

Choosing powers of 2 for the b_i's might lead to hash functions that are quicker to compute using bitwise operations. However, rounding up/down to powers of 2 makes the hashing less balanced among the attributes, which contributes indirectly to overall performance by requiring more tuples to be scanned in the level-2 store. Further, for most queries the hashing cost will be small compared to the scanning/aggregation cost in the level-2 store.

```
initialize(total); /* for sum, would be "total=0" */
for ( x1=1; x1<=b1; x1++ ) {
     for ( x2=1; x2<=b2; x2++ ) {
           for each tuple in the list starting at S[x1][x2][y3]...[y8] {
                   if the tuple matches the query
                          accumulate(total,aggregate-value);
                   /* for sum, would be "total += aggregate-value" */
           }
     }
}
```

Figure 2. Pseudo-code for computing tuples from the level-2 store

Choosing the h_i Functions

The properties desirable for the h_i functions are that they should uniformly distribute tuples among the range $1, \ldots, b_i$ as much as possible. By distributing tuples close to uniformly, we avoid the problem of having some very long lists and many empty lists in the slot array: the empty lists would hardly be used by queries, and the long lists would be traversed by many independent queries, leading to poor performance. Like any hash function, there is a limit to how good a job h_i can do in the presence of skew. If some value of attribute i occurs fifty percent of the time, then the image of that attribute under h_i must occur at least fifty percent of the time.

We can be a little more informed than an ordinary hash function, because we can precompute the single-attribute distributions in advance, and adjust the h_i functions accordingly. Consider a domain of size N for attribute i, with $N \geq b_i$ and let $f_1, \ldots, f_N$ be the frequencies of each of the N attribute values. We wish to partition the domain into subsets $S_1, \ldots, S_{b_i}$ such that each S_j is of roughly the same cardinality, and so that $\sum_{x \in S_j} f_x$ is as balanced as possible.

By choosing S_js of the same cardinality, we reduce the number of bits needed to distinguish members of each S_j. By balancing the frequencies, we make the mapping into the slot array more uniform. An exact solution to this subproblem is beyond the scope of this paper. A useful heuristic is to allocate domain elements to sets S_j in decreasing frequency order, putting a domain element in the set with lowest total frequency, stopping when the cardinality of S_j is larger that N/b_i.

Some b_i values may be slightly higher than others: some will equal $\lfloor T^{1/d} \rfloor$ and others will equal $\lceil T^{1/d} \rceil$. We can analyze each of the attributes to see which would benefit the most (in terms of spreading the data more uniformly) and choose those attributes as recipients of the higher b_i values. Once the mappings to sets S_j has been determined, our hash functions can be implemented by storing each (domain-value,j) pair (for a domain value mapping to S_j) itself in a hash table or sorted array.

Optimizing the Size of the Slot Array

It seems that there should be a natural size for the slot array. A slot array that was too big would give poor time performance because too many (mostly empty) slots would need to be checked. A slot array that was too small would give poor time performance because most lists would be very long, with few actual matches, and time would be wasted traversing the lists.

Thus we seek the time-optimal slot array size. Since queries with the most ALLs will dominate the average cost, we can approximate the overall time-optimum by optimizing the slot array size for queries at the first level not substantially present in the level-1 store; call this the "critical level". Let k denote the number of ALLs at the critical level. In the following analysis, we assume a perfectly balanced distribution of b_i values, and a uniform distribution of tuples among lists.

If T is the slot array size, then we will examine $T^{k/d}$ slots for a critical-level query. Thus the overall cost is equal to

$$T^{k/d}(s + nl/T)$$

where n is the number of finest granularity tuples, s is the cost of a slot access, and l is the cost of a list element access. This function is minimized when

$$T = \frac{nl(d-k)}{ks}$$

So, for example, if $k = d/2$, and $l = s$, we would expect a number of slots equal to the number of finest-granularity tuples.

So far we haven't considered any limitations on memory availability. As we'll see in Section 3.5 we will probably settle for a slot array size smaller than the time-optimal size when the memory budget is limited. We may not have sufficient memory for a time-optimal choice. Further, it may be more beneficial to spend memory on the level-1 store than to spend it on reaching the time-optimal slot array size.

3.5 Space Issues and Tradeoffs

We will clearly have a limited space budget, and so we need to specify how to allocate that space among the level-1 and level-2 stores.

List Representation in the Level-2 Store

The list representation can be made compact by observing that we don't need to store complete key information in the list nodes. We already have partial information about the key based on the slot that the list emanates from. For attribute i, if g_i is the maximum number of elements that map to a single element via h_i, then we need to store a number in the range $1, \ldots, g_i$ for attribute i. Thus we need to store a number in the range $1, \ldots, g_1 g_2 \cdots g_d$ in each list element. Each list element also needs a pointer to the next list element.[3]

Given the size p of a pointer in bits, we can estimate the total size of the lists in the level-2 store as $n(p + \log(g_1 g_2 \cdots g_d))$ bits. To make the matching more efficient within a list, using bit operations typically found in machine instruction sets, we could specify separate bit ranges for each attribute. In that case the space needed is $n(p + \lceil log(g_1) \rceil + \ldots + \lceil log(g_d) \rceil)$

Assuming that the finest-granularity data fits in memory, this amount is smaller than the available memory. The remaining memory is used for the level-1 store and the slot array in the level-2 store.

Space in the Level-2 Slot Array

Consider a datacube tuple t from cuboid i with k_i ALLs. The cost of looking in the level-2 store for t is $T^{k_i/d}(s + nl/T)$. Thus the average cost of a lookup to the level-2 store is given by

$$A = \sum_{t \text{ not in level-1 store; all cuboids } i} t_{i,j} T^{k_i/d}(s + nl/T)$$

where $t_{i,j}$ is the probability that t is queried. Suppose we keep a running sum of the $t_{i,j}$ values of *unmaterialized* tuples at each value of i in the level-1 store, and denote the sum as C_i for $i = 1, \ldots, 2^d$. Then the equation above reduces to

$$A = \sum_{i=1}^{2^d} C_i T^{k_i/d}(s + nl/T).$$

[3] One could optimize the list so that a node contains more than one element, saving some space for pointers, but we do not pursue such optimizations here.

The derivative of A with respect to T is given by

$$A' = \sum_{i=1}^{2^d} C_i (k_i s T^{k_i/d - 1} - nl(d - k_i) T^{k_i/d - 2})/d.$$

We will use these expressions to evaluate the tradeoff between the level-1 and level-2 stores below.

Space in the Level-1 Store

Suppose that we've computed the datacube, and are trying to decide which tuples to place in the level-1 store. Materializing a tuple t from cuboid i with k_i ALLs and a probability of $t_{i,j}$ will benefit us by an amount equal to

$$B_t = t_{i,j} T^{k_i/d}(s + nl/T).$$

This amount corresponds to the expected reduction in cost of searching the level-2 store.

Thus we should materialize datacube tuples in strictly decreasing order of their B_t values. Observe that the formula for B_t is exponential in k_i while linear in $t_{i,j}$. We would materialize a level $k-1$ tuple in preference to a level k tuple only if the probability of the level $k-1$ tuple was (roughly) a factor of $T^{1/d}$ higher than that of the level k tuple. As a result, our materialization pattern will likely include a good coverage of the top levels of the cube, with much more scattered coverage of the remainder of the cube.

The Tradeoff Equation

We can measure the cost of storing one level-1 tuple as the product of the tuple size by a hash fudge factor (typically 1.2). Let's suppose that the storage cost per tuple at level-1 is given by q. The storage cost of one slot in the level-2 store is equal to the slot size. Let's denote the slot size by z.

We have some memory left over after accounting for the lists, and we need to decide how to balance these needs among level-1 storage (materializing additional tuples) and level-2 storage (increasing the size T of the slot array). Our approach is to give a unit of memory to the piece of storage the "needs" it the most. We measure the need of the level-2 store as $-A'/z$. We measure the need of the level-1 store as B_t/q where t is the next tuple in line to be materialized. Which of these quantities is greater determines which store will be allocated the next unit of storage.

After such an allocation, the parameters change, and B_t and A' need to be recalculated. We continue this process until we have exhausted all of the available memory. (In practice, we could save computation by allocating memory in batches, rather than one unit at a time.)

3.6 Computing the Materialized Tuples

We can use a datacube computation algorithm [1, 4, 12, 15], to materialize datacube tuples on disk. In this cube computation, we compute both the original aggregate as well as any additional aggregates (such as the count) needed to compute the tuple probabilities. We materialize the cube on disk in d files, one for each number of ALLs. An external sort is performed on each of these files to sort them into decreasing probability order.

The finest-granularity file gives us n, the total number of finest-granularity tuples, which we need in order to configure the stores appropriately. We can also calculate the number of distinct values of each attribute, and their frequencies from the tuples with $d-1$ ALL's. This information will help us choose the b_i values and construct the h_i functions.

We then treat each of the d files as a sorted run to be "merged" according to the formula for B_t (Section 3.5). Tuples are read and inserted into the level-1 store until all of the memory is accounted for. (We could use an extendible hashing technique such as linear hashing [9] to avoid having to rebuild the hash table as more tuples are read.)

In the case of workload-based distributions, we need to also consider the probabilities of tuples at each granularity that are *not* in the datacube result. In general, we would have to explicitly list all possible combinations of attributes that yielded a nonzero probability, and consider them for materialization in the level-1 store. However, for full granularity distributions, there is a particularly efficient solution that does not require the storage of nonexistent tuples.

Recall that in a full granularity distribution, all combinations of non-ALL attributes at a particular granularity are equally likely. As a result, once one tuple at that granularity is chosen for materialization, all of the remaining tuples at that granularity must be next in line for materialization, according to our formula for B_t. Thus, given sufficient space to materialize the whole granularity, the complete cuboid will be materialized. Rather than explicitly materializing the nonexistent tuples, we can simply switch on a bit indicating that the granularity has been *completely* stored in the level-1 store. Thus, if the bit is on and a query to that granularity finds no match in the level-1 store, we can immediately conclude that the tuple doesn't exist, without consulting the level-2 store. The overhead (2^d bits) is negligible.

At this point we have derived a value for T, the slot array size. We allocate the slot array of size T, derive corresponding balanced b_i values, and construct the h_i functions. Finally, we read in the finest granularity data and store it in the level-2 store.

3.7 Maintaining Materialized Tuples

In our framework, we may have new data being frequently appended to the existing data. In such a situation, we have to update or add a tuple to our level-2 store as well as update any materialized tuples in the level-1 store.

If the tuple t has a matching tuple in the level-2 store, we update its value. Otherwise we add this tuple to the level-2 store. Note that a new tuple will map to exactly one slot, so we traverse just one list on an update.

In general, we may need to check 2^d slots in the level-1 store for all possible tuples that might need to be updated. For a count based distribution, however, if a tuple t is materialized in the level-1 store, any generalization [13] (the tuple formed by replacing one or more attribute values by ALL's) of the tuple has at least as high a count and more ALL attributes, and so must also be present in the level-1 store. When a tuple t is added, the first tuple we update is the tuple with ALL in every attribute position. We then search for generalizations of the tuple on a level by level basis starting from the level of the cuboid where all but one attribute is ALL. If a tuple is not found, we know that no specialization of this tuple would also be found. This breadth first traversal of the cube allows us to cut down on the 2^d possible cube updates.

Another potential optimization of the update process involves keeping a bit for each cuboid indicating whether there is at least one tuple from that cuboid in the level-1 store. One can then cheaply exclude from the 2^d cuboids any cuboid that has no tuples materialized. (We shall see experimentally that this optimization improves update times.)

If the number of appended tuples is large we may need to periodically rebuild our data structures for one of several reasons:

1. There is no space remaining in memory.
2. The underlying data distribution and hence the best choice of h_i has changed.
3. The number of slots may no longer be ideal since the average number of tuples in a slot becomes too large.
4. New domain values appear for some attributes.

We would not expect this reorganization to take place very often, since the base data contributing to the stored datacube is probably orders of magnitude larger than the size increments accumulated over a short period. To plan for future updates we could allow some spare memory for new records, and preallocate some extra attribute domain space for yet-unseen attribute values.

4 Extensions

4.1 Range Queries

We can extend our framework to efficiently answer range queries. A range query specifies bounds on attribute values rather than specifying an ALL. The simplistic way of answering a range query would be to treat a range query as the equivalent query with ALL's over the level-2 store, and then check if each matching tuple falls within the range. Essentially we are *pulling up* the selection.

We adopt a more sophisticated approach where we construct the h_i functions such that they preserve the domain ordering on the attributes. In other words, if x and y were values in the domain of attribute i such that $x < y$ in the domain ordering, then it must be the case that $h_i(x) \leq h_i(y)$. Then to answer a query in the level-2 store for attribute i between x and y, we need only check offsets between $h_i(x)$ and $h_i(y)$ in the ith index of the slot array. Our hash function h_i now performs a special kind of hashing known as range-partitioning.

Example 4.1: Consider a modification of Example 3.1 in which the second attribute is used in a range constraint instead of as an ALL. Suppose that we wish to compute the aggregate for values of the second attribute between v_2 and v_2'. Let `low2` equal $h_2(v_2)$ and `high2` equal $h_2(v_2')$. Then the query would be answered using the pseudo-code in Figure 3 (using the same terminology as Example 3.1). We traverse `b1*(high2 - low2 + 1)` lists. □

The cost of such an approach is that the best h_i functions obtainable under the ordering constraint may be less uniform than an unconstrained choice.

A similar technique could be used in the level-1 store if we use a hash function that is similar to the composition of the h_is in the level-2 store. We would need to be certain that all tuples in the query's domain (i.e., with the right number of ALLs) actually appear in the level-1 store. Thus, to support range queries, we have an extra incentive to materialize *all* tuples in a datacube level, rather than leaving out some tuples with very small counts.

4.2 Hierarchies

In real life we frequently have hierarchies on attributes. For example, in a dataset on the American National Basketball Association the teams fall into a natural hierarchy (Figure 4). In general, the hierarchy may not be a balanced tree.

One branch of this hierarchy would be `Knicks`, `Atlantic Division`, `Eastern Conference` specifying the team name, division and conference.

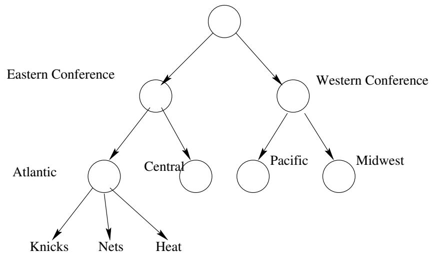

Figure 4. A partial hierarchy for the NBA

While one user may be interested in calculating the total points scored at the division level, another might be interested in the same statistic at the conference level. In our base data, we would only be storing the information about the team; it would be a waste of space to store the division and conference since these are dependent upon the team.

We can adapt our scheme to handle hierarchies by mirroring the structure of the hierarchy in our domain ordering. We order our domain lexicographically from higher levels in the hierarchy down to the bottom level in the hierarchy. In the example above, we order the teams according to (conference,division,team). Queries on the hierarchy now correspond to range queries, which we would handle as in Section 4.1.

5 Experimental Results

We experiment with an 8 dimensional subset of cloud coverage data [6] which has 1015367 base tuples. The total number of datacube tuples is 102745662. Each tuple can be represented by 64 bits for the CUBE BY attributes and 32 bits for the aggregate. The finest-granularity data would occupy 12 megabytes, while the full datacube would occupy 1.2 gigabytes.

We also carry out experiments on a uniform dataset. This uniform dataset has the same dimensionality and number of tuples as the cloud coverage data. The cardinality of the attributes are fixed to be the same as those of the cloud data. This enables us to compare results across both datasets. Though the base data contains 1015367 tuples, the datacube itself is larger and consists of 195206797 tuples.

We have implemented the algorithms presented here in `C` and the experiments were carried out on a 300 MHz Sun Ultra-2 running Solaris 5.6. The implementation consists of three distinct modules. The first module analyzes the dataset and computes the optimal size of the level-1 and level-2 stores. The second module chooses the hash functions and bucket sizes for each of the attributes. The third module is the core of the system where any datacube query is answered using the level-1 or the level-2 store. The timing measurements are computed by measuring the total

```
initialize(total); /* for sum, would be "total=0" */
for ( x1=1; x1<=b1; x1++ ) {
     for ( x2=low2; x2<=high2; x2++ ) {
           for each tuple in the list starting at S[x1][x2][y3]...[y8] {
                   if the tuple matches the query
                           accumulate(total,aggregate-value);
                   /* for sum, would be "total += aggregate-value" */
           }
     }
}
```

Figure 3. Pseudo-code for Range Queries

execution time taken for a set of queries and averaging over the size of the set. In the following graphs the horizontal axis shows the number of megabytes available beyond the space needed for the finest-granularity data.

Example 5.1: Consider the cloud coverage dataset. Figure 5 shows various measures of the performance of our algorithm for a range of available memories. Figure 5(a) shows the chosen size of the slot array in megabytes. Figure 5(b) shows the space required for the level-1 store in megabytes. Figure 5(c) shows the overall query cost to the level-2 store. (The cost of checking the level-1 store is negligible.) Figure 5(d) shows the number of tuples from each level in the cube that are resident in the level-1 store. Figure 5(e) shows the percentage of tuples of each level in the cube (the number of tuples from a level present in the level-1 store divided by the total number of tuples present in that level in the complete cube) resident in the level-1 store. Figure 5(f) shows the average update cost of a tuple for two different strategies. The first strategy checks for each of the 2^d generalizations of a tuple in the level-1 store while the second strategy only looks for generalizations for which at least one tuple has been materialized (as described in Section 3.7). The queries are generated according to the count based distribution.

These results show that as we increase memory there is a rapid increase in the number of slots which subsequently tapers off. The graph is flat in some places because there is no distinct integer solution which corresponds to the predicted slot array size. In contrast the number of tuples in the level-1 store increases linearly with memory. There is initially a rapid decrease in query cost. As we increase the amount of memory available the decrease becomes less rapid. The initial gain is very sharp since we are first materializing the very expensive tuples with a high number of ALL's. Figure 5(d) shows that we do not proceed strictly in a level by level fashion while populating the level-1 store. Figure 5(e) illustrates the impact of materializing the high count tuples; the normalized counts of tuples stabilize quicker than the number of tuples in the previous figure. Figure 5(f) shows that checking the bit for cuboids with no materialized tuples allows us to save greatly on the update cost when the size of the level-1 store is small. □

Example 5.2: Consider the uniform dataset with a count based query distribution. Figure 6 shows various measures of the performance of our algorithm for a range of available memories. Figure 6(a) shows the chosen size of the slot array in megabytes. Figure 6(b) shows the space required for the level-1 store in megabytes. Figure 6(c) shows the overall query cost to the level-2 store. Figure 6(d) shows the number of tuples from each level in the cube that are resident in the level-1 store. Figure 6(e) shows the percentage of tuples from each level in the cube that are resident in the level-1 store. Figure 6(f) shows the average update cost of a tuple for two different different strategies outlined earlier.

The results for the slot array size, number of tuples in the level-1 store and the expected query cost are similar to those of the cloud dataset. As indicated in Figure 6(d) and Figure 6(e), for uniform datasets, we materialize tuples from one level only after materializing a considerable fraction of the earlier levels. Each level has a distinct starting point on the X axis (we do not reach the starting points for some of the levels) unlike the cloud coverage dataset where we materialize the high benefit tuples from each of the level at the beginning. For uniform data we tend to materialize whole cuboids (since all tuples in a cuboid have counts close to each other) which is reflected in the fact that the curves are less smooth. Figure 6(f) shows that the difference between strategies is significant even when the size of the level-1 store is large. This is because even with a large level-1 store many cuboids are still untouched. □

Some interesting features of our results are the tradeoffs between the slot array size and that of the level-1 store. Initially a slight increase in the slot array size causes a great reduction in the partial match cost (the average number of list elements is greatly reduced), with large amounts of memory the slot-array tends to its natural size. The

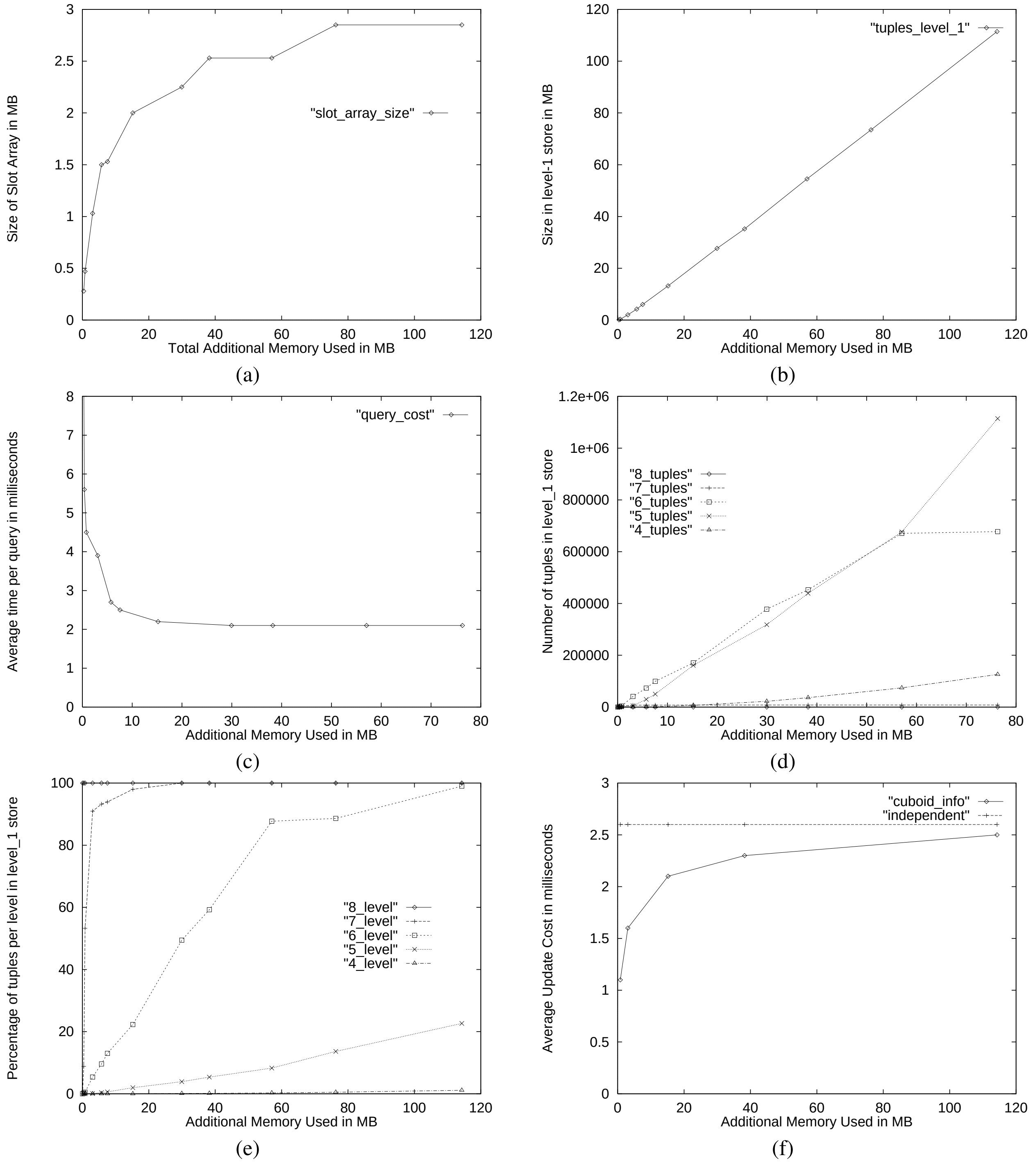

Figure 5. Experimental results for Example 5.1.

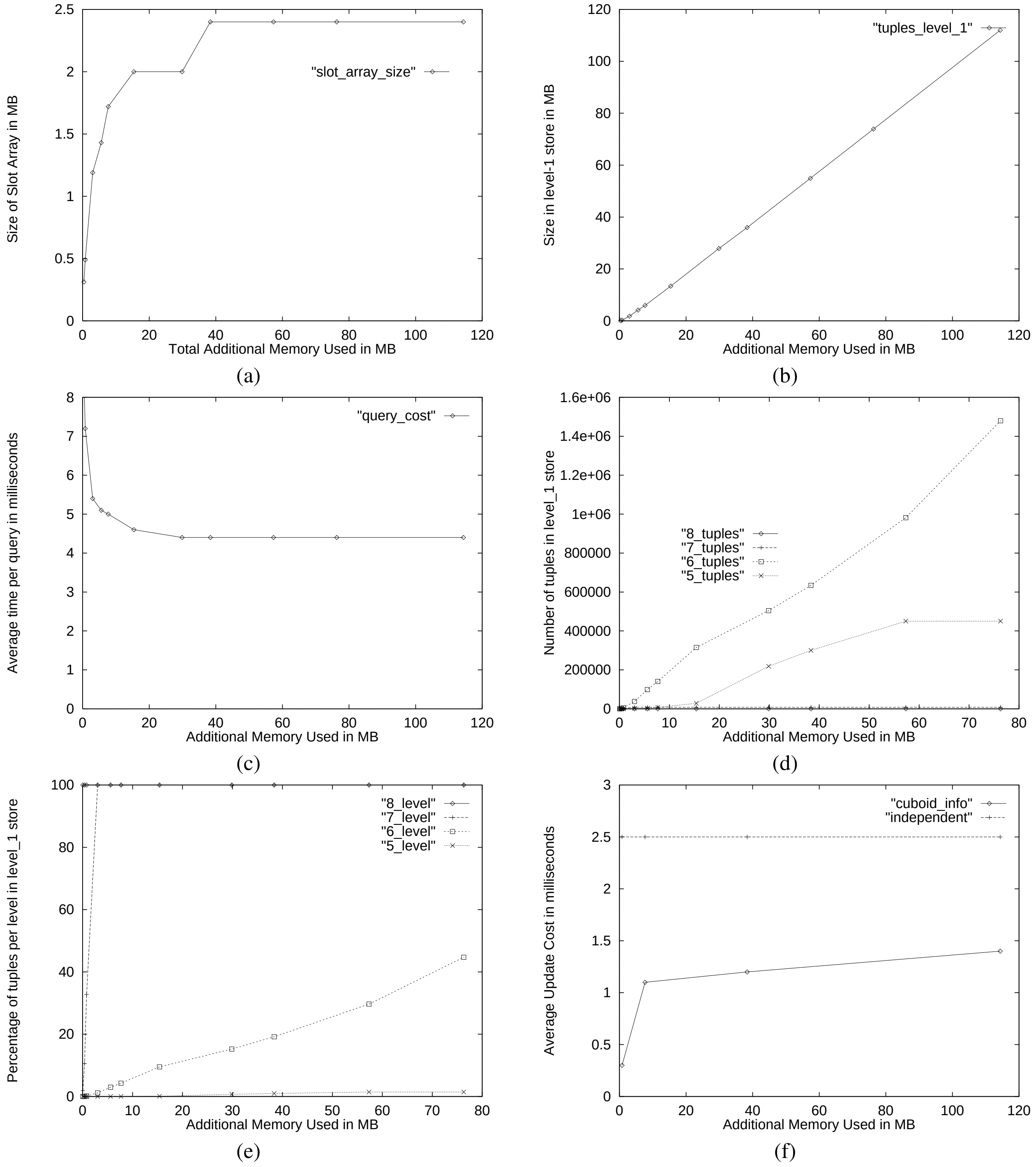

Figure 6. Experimental results for Example 5.2.

difference between the uniform and real world data in terms of probability distribution within a level illustrates that for non-uniform datasets it is profitable to choose the tuple over the cuboid as the unit of materialization.

Example 5.3: We now experiment with the cloud coverage dataset according to the uniform cuboid query model. The graphs are very similar to Example 5.1, and are omitted here. The sizes of the level-2 store and the level-1 vary in a similar fashion to the graphs of the high count query model. The main distinction lies in the composition of tuples materialized in the level-1 store. Since each tuple in a cuboid has the same benefit, the effective unit of materialization becomes a cuboid rather than a tuple. All the tuples with 7 ALLS are materialized rapidly while no tuples with 4 ALLS are materialized across the entire range of additional memory. This levelwise progression is characteristic of this query model. □

In Figure 7 we show the number of buckets assigned to each attribute of the cloud dataset for a estimated level-2 size of 90000 slots. The solution which assigns 86400 slots tries to balance the b_i values, but uses some additional heuristics. For example, any attribute that has a single value occurring more than fifty percent of the time is assigned just two buckets.

Once the b_i values for each of the functions have been assigned we use the methods described in Section 3.4 to compute the hash functions themselves. The images of each attribute value for attribute number 1 in the cloud data set are shown in Figure 8. The above techniques are used for computing the b_i's and h_i's in all the previously described experiments.

We see that for all the examples, with just tens of megabytes of additional memory, our response time is between 2 and 4 ms. The time taken to compute a tuple by scanning an array containing the finest granularity data would have been 194 ms. Compared with a disk based approach, our techniques provide query performance faster than a disk lookup on a conventional disk with latencies of at least 10 ms per access.

While we clearly beat disk-based systems for single datacube tuple queries, we may not beat a disk-based materialization for some queries returning sets of tuples. For example, consider a dataset with three dimensions, namely state, year, and model, for a car sales database. A query such as "Find total sales for each combination of year and model in New York" would be particularly efficient in a disk-based system if the data were clustered lexicographically by (state,year,model). The answers would reside on a small number of disk pages, so the I/O cost could be amortized over many tuples. Our approach, on the other hand, would pay a computation cost for each answer tuple, which could add up to more than the I/O cost.

Attribute Number	Cardinality	Buckets Assigned
1	30	6
2	8	4
3	152	5
4	352	5
5	7037	6
6	101	2
7	10	2
8	179	6

Figure 7. Number of buckets assigned per attribute

Attribute Value	Hash Image
1, 10, 11, 23, 29	1
5, 6, 16, 17, 26	2
7, 8, 9, 15, 18	3
12, 19, 24, 27, 28	4
2, 13, 14, 21, 25	5
3, 4, 20, 22, 30	6

Figure 8. Hash function for attribute 1

Despite the scenario above, we still expect to beat disk-based materializations for most queries returning sets of tuples. If the query above had instead been "Find total sales for each combination of state and year for Taurus cars," then the clustering by (state,year,model) would not help, and many more disk pages would need to be accessed. In the absence of redundancy,[4] there can be just one physical organization. That organization will benefit some queries, but many queries will still require essentially random I/O.

6 Related Work

Most related work [7, 14] has focussed on the problem of materializing cuboids on disk with a constraint on the amount of disk space available. In [8] the problem of efficiently storing and querying the materialized tuples is addressed by using *cubetrees*. An alternative to materialized datacubes called small materialized aggregates (SMA) is introduced in [10].

In [5], views and indices are chosen to be materialized in terms of a set of queries Q. The algorithms introduced operate upon a data structure called a query view graph which contains nodes corresponding to each query as well as each possible view. Note that the total number of possible queries over a datacube is even larger than the datacube

[4] Redundancy is not a good option, because of the size of the datacube and the number of copies needed to cover all attribute prefixes.

itself. Hence, data structures like a query view graph are not scalable for a large set of queries.

There are a number of different datacube computation algorithms. The array-based algorithm proposed by Gray et al. [4] is essentially a main memory algorithm, where all the tuples of the finest level of the datacube are kept in memory as a k dimensional array, where k is the number of `CUBE BY` attributes. Zhou et al. [15] have proposed an array based algorithm that computes the datacube using array-chunking techniques. By managing the order in which chunks are processed, substantially less of the result array needs to be kept in memory at any one time than with the algorithm of Gray et al. The PIPESORT algorithm [1] tries to optimize the overall computation of a datacube by providing a set of paths which cover the search lattice and then executing each path in turn. This algorithm makes use of cost estimates of the various ways to compute each cuboid to determine which parent cuboid will actually be used to compute the tuples. The OVERLAP algorithm [1] tries to minimize the number of disk accesses by overlapping the computation of the cuboids, by making use of the partially matching sort orders to reduce the number of sorting steps performed. `Partitioned-Cube` and `Memory-Cube` [12] are efficient algorithms for computing datacubes which work particularly well for sparse data.

Most typical partial match retrieval schemes [2, 11] are hash based. In these schemes each attribute in a record maps to a bitstring and the we obtain a signature by concatenating the bitstrings. Each record is mapped into a bucket based on its signature. A partial match query is processed by producing the bitstrings for the specified attributes. Based on these bitstrings we can compute a set of candidate signatures by filling in the unspecified bits in all possible ways. We check the buckets corresponding to each candidate signature for matching records. These schemes optimize the number of bits assigned to the bitstring of an attribute based on the probability of that attribute being specified in a query. Our technique is more efficient than partitioning by bits since we are not limited to b_i values being powers of 2.

Our work is distinctive in that we focus on making appropriate use of available main memory. Our unit of materialization is a tuple rather than cuboid, a decision that is appropriate when dealing with skewed rather than uniformly distributed data. We also investigate how to best organize the finest granularity cuboid tuples to best answer queries which require an unmaterialized datacube tuple as the answer.

7 Conclusions

We have introduced a main memory based framework for answering datacube queries efficiently. We exploit the fact that while the entire datacube is much larger than available memory, there is often enough space for the finest granularity cuboid. By materializing a well chosen set of tuples in memory, it is possible to compute any datacube tuple efficiently without having to go to disk. Apart from the reduced time for processing queries this also requires considerably less maintenance cost than a disk based scheme in an append-only environment. Our experimental results show query performance in the 2-4 ms range for a typical example. Future work will consider how to deal with situations where the finest granularity data doesn't fit in memory.

References

[1] S. Agarwal et al. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International VLDB Conference*, Mumbai,India, 1996.

[2] A. Aho and J. Ullman. Optimal partial-match retrieval where fields are independently specified. *ACM Transactions on Database Systems*, 4(2):168–179, 1979.

[3] P. Bernstein et al. The Asilomar report on database research. *ACM Sigmod Record*, 27(4), 1998.

[4] J. Gray et al. Data Cube: A relational aggregation operator generalizing group-by, cross-tab and sub-total. *Data Mining and Knowledge Discovery*, 1(1), 1998.

[5] H. Gupta et al. Index selection for OLAP. In *Proceedings of the 13th ICDE*, 1997.

[6] C. J. Hahn et al. Edited synoptic cloud reports from ships and land stations over the globe, 1982-1991. Available from http://cdiac.esd.ornl.gov/cdiac/ndps/ndp026b.html, 1994.

[7] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD Conference*.

[8] Y. Kotidis and N. Roussopoulos. An alternative storage organization for ROLAP aggregate views based on cubetrees. In *Proceedings of the 1998 ACM SIGMOD Conference*.

[9] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Databases*, 1980.

[10] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proceedings of the 24th International VLDB Conference*, 1998.

[11] K. Ramamohanrao, J. Lloyd, and J. Thom. Partial-match retrieval using hashing and descriptors. *ACM Transactions on Database Systems*, 8(4):552–576, 1983.

[12] K. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 23rd International VLDB Conference*, 1997.

[13] K. Ross and K. Zaman. Optimizing selections over data cubes. Technical Report CUCS-011-98, Department of Computer Science, Columbia University, 1998.

[14] A. Shukla, P. Deshpande, and J. Naughton. Materialized view selection for multidimensional datasets. In *Proceedings of the 24th International VLDB Conference*, 1998.

[15] Y. Zhou et al. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD Conference*.