Efficient Implementation of Large-Scale Multi-Structural Databases

R. Fagin Ph. Kolaitis R. Kumar J. Novak D. Sivakumar A. Tomkins

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120

Abstract

In earlier work, we defined "multi-structural databases," a data model to support efficient analysis of large, complex data sets over multiple numerical and hierarchical dimensions. We defined three types of queries over this data model, each of which required solving an optimization problem. An example is to find the ten most significant non-overlapping regions of geography crossed with time in which coverage of the Olympics was much stronger in newspapers than online sources.

In this paper, we present a general query framework capturing the original three queries as part of a much broader family. We then give efficient algorithms for particular subclasses of this family. Finally, we describe an implementation of these algorithms that operates on a collection of several billion web documents. Using our algorithms in conjunction with random sampling techniques, our system can solve these queries in real time.

1 Introduction

Massive repositories of structured, semi-structured, and unstructured data are growing in prevalence and importance. However, the query languages we use to address these corpora focus largely on relevance ranking, in the case of unstructured data, and various types of aggregates in the case of structured data. An important task for database research is the development of frameworks that support rich analytical queries through a coherent and principled architecture, rather than an assortment of ad hoc solutions.

Towards this goal, Fagin et al. [1] have recently introduced a theoretical framework called the *multi-structural database* (or *MSDB*). As our work lies within this framework, we begin with a brief description of its salient characteristics using a running example of a database of newspaper articles.

(1) The *schema* of an MSDB is a collection of dimensions, each of which is a bounded lattice. Subsets of these dimensions may be mechanically combined to form a new multi-dimensional lattice, whose elements represent multi-dimensional restrictions of the data objects.

In the example of newspaper articles, consider a schema containing two dimensions: locations the article is relevant to, and the article's date of publication. The first of these dimensions, `geography`, admits a natural hierarchy (continents, countries, cities, etc.), where natural restrictions correspond to nodes of the hierarchy; we may thus speak of documents relevant to Europe, or to France in particular. The second dimension, `date`, is a numerical attribute, and natural restrictions are obtained by considering time intervals. By combining these two dimensions, we generate restrictions such as "articles relevant to Europe from the first half of 2003."

(2) A *pairwise-disjoint collection* (abbreviated PDC) is a set of restrictions (lattice elements) that are conceptually non-overlapping in the sense that every pair in a PDC is non-overlapping in at least one dimension.

The restrictions "articles about Europe from the first half of 2003," "articles about California from June 2004," and "articles about France dated 6/17/2004–9/23/2004" are pairwise disjoint. The first two are disjoint in both the `geography` and `date` dimensions; the latter two are disjoint in the `geography` dimension even though they overlap in the `date` dimension; and, the first and the third restrictions are disjoint in the `date` dimension even though France is a descendant of Europe in the `geography` hierarchy.

We say the collection is *conceptually* non-overlapping because this is a property of the schema, not the data. There may be certain documents that are relevant to both France and Spain, and hence two conceptually non-overlapping regions may in fact contain a common article.

(3) The *data* in a multi-structural database consist of a collection of data objects, together with a membership re-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

lation that describes the lattice elements to which each data object belongs.

The membership relation for the newspaper articles simply specifies, for each article, the most specific locations that it is relevant to (namely leaves corresponding to cities, internal nodes corresponding to countries, etc., in the `geography` hierarchy), and the calendar date when it was published.

(4) A *query* is an optimization problem that takes as input a collection of data objects, and returns a PDC of a given size (that is, a family of conceptually-disjoint restrictions of the objects). The particular PDC returned must optimize some objective function; this objective function encodes the particular query. Since the intent of analytical queries is to seek an understanding of trends or patterns in the data, rather than obtain exact answers, good approximations are often acceptable if they are efficiently computable.

As an example, consider asking for the ten time intervals that exhibit the maximum contrast in the number of articles about Europe versus the number of articles about Canada.

Our contributions. The present paper advances the multi-structural database framework in three fronts. First, we provide a conceptual setting for formulating and analyzing queries in a principled manner. Second, we give highly efficient algorithms (a linear time algorithm for a very important schema family and an indexing/sampling methodology). And third, we demonstrate the richness of the framework and the quality of the algorithms by applying them to a massive, petabyte-scale collection of documents and measuring the results.

Our first contribution is conceptual in nature. Our earlier work in [1] renders three specific analytical operations (DIVIDE, DIFFERENTIATE, DISCOVER) in the form of optimization problems. In this paper, we present a rich family of "decomposable" objective functions that operate by examining and scoring the data objects belonging to each restriction of the PDC in turn, and then mechanically combining these scores. We show that, for combinations of hierarchical and numerical dimensions, all such optimization problems can be solved (perhaps approximately) in polynomial time. The three operations described above are of this form.

Our second contribution is algorithmic in nature. We describe two important subclasses of the class of decomposable objective. First, the class of *min-monotone* functions maximizes the minimum over all elements of the PDC of a monotone function applied to that element. Second, the class of *sum-additive* functions maximizes the sum over all elements of the PDC of an additive function (in a sense to be defined later) applied to the element. We give more efficient algorithms for both of these classes.

Sum-additive functions arises very naturally in several contexts, especially for numerical dimensions (time, salary, sales volume), where one seeks to find k intervals of highest total value of a function that is additive over disjoint sub-intervals (number of articles, wages paid, cans of beer sold). In fact, DIFFERENTIATE and DISCOVER, two of the three analytical operations proposed in [1], are of this kind; an example of a DIFFERENTIATE operation is to find the k disjoint time intervals that have the highest total contrast in the number of articles about Europe and articles about Canada. Our technically most intricate contribution is an exact optimization algorithm for sum-additive objective functions that runs in time linear in the number of data objects: this is a significant improvement over the previous quadratic algorithm for numerical dimensions [1], particularly since the number of objects is often quite large. Underlying this result is an efficient algorithm for the following basic combinatorial problem: given a sequence of integers, find k non-overlapping contiguous subsequences whose sum is maximal.

Abstracting the objective function in terms of a combination of scores over each element of a PDC has an additional bonus from the viewpoint of efficiency. Namely, it is often possible to obtain unbiased estimators for the objective function using random sampling. This feature is crucial in our implementation, since when dealing with massive data sets, it may be too expensive to even scan all the data objects necessary to compute the "quality" of various restrictions; for this reason, we need to rely on randomly sampling data objects corresponding to each of the restrictions.

Finally, our third main contribution is a demonstration of the MSDB framework on large-scale real-world data. A prototype implementation of an MSDB system with three analytical operators (DIVIDE, DIFFERENTIATE, DISCOVER) is described in [1], with experiments dealing with a few thousand data objects. Here, we present an implementation of an MSDB of nearly four billion web documents (10^{14} bytes; 10^{15} bytes including metadata) from the WebFountain project at IBM [3]. The architecture of our implementation consists of a back-end that offers the ability to materialize any required restriction of a collection of data objects, and a front-end query engine that produces restrictions as needed in the optimization of a desired objective function.

In our experiments, we measure the time required for the WebFountain backend to produce the necessary data, and then separately measure the time required for the multi-structural query engine to find the optimal PDC, once this data has been made available. Our primary goal is not to compare different approaches to this problem, although we do so where possible, but to argue that multi-structural optimization queries may be computed via random sampling in real time over corpora consisting of billions of data objects.

We consider six families of analytical operations, three from [1] and three new ones. We then create a benchmark set of 33 queries, each of which is a concrete instantiation of one of the six families. We present measurements showing that our unoptimized system is capable of returning answers for most queries in between one and ten seconds depending on the required accuracy, while a few more complex queries take no more than a few minutes to compute.

The remainder of the paper proceeds as follows. In Section 3, we outline the theoretical underpinnings of the framework. In Section 4, we give a detailed presentation of the new algorithm for contiguous subsequences, together with some actual runtimes comparing the new algorithm to previous approaches, and some examples of the algorithm in use. Section 5 describes the architecture of the MSDB system we have implemented. Section 6 describes some experimental results.

2 Related work

There is a vast literature that we will not attempt to summarize here on multidimensional data models and on-line analytical processing (OLAP) queries. Comprehensive overviews of several different topics on multidimensional databases can be found in the book [7]. In particular, Torlone's chapter [8] in that book gives an account of various models for multidimensional data. Our lattice-theoretic modeling of multidimensionality is not new. Indeed, lattices have been considered in earlier work on multidimensional data, as they naturally generalize tree-structured hierarchies and have the advantage of being closed under the direct product operation. Our formulation is reminiscent of that of Harinarayan, Rajaraman, and Ullman [4], who use lattices (and direct products of lattices) as first-class citizens to model multiple, hierarchical dimensions. Lattices also underlie various other formulations of dimensions and hierarchies, including those in [5, 6, 9].

See [1] for a detailed discussion of differences between our work and standard OLAP. The key distinguishing feature of our work compared to earlier work on OLAP is the class of queries that can be expressed and answered in our framework. While typical OLAP queries ask for aggregate or summary results along specified points in a multi-dimensional lattice, our framework supports powerful queries expressing optimization problems. Furthermore, an answer to such a query is a pairwise disjoint collection (PDC) of dimensions that maximizes a certain "measure". This makes it possible to formulate a variety of clustering, trend-discovery, and hierarchy-aware summarization problems in a unifying framework.

3 Framework

A *multi-structural database* (or simply *MSDB*) (X, D, R) consists of a universe $X = \{x_1, \ldots, x_n\}$ of *objects*, a set $D = \{D_1, \ldots, D_m\}$ of *dimensions*, and a *membership relation* $R \subseteq X \times V$, where $V = \cup_i D_i$.

We will treat each x_i as simply an identifier, with the understanding that this identifier may reference arbitrary additional data or metadata, such as the content of the corresponding document. Each dimension D_i is a bounded lattice[1]. This formulation represents a generalization of numerical and hierarchical dimensions, but for the purposes of this paper, the reader may think of the dimensions as being either hierarchical or numerical. We assume that the lattice nodes used in all lattices are distinct; the vocabulary $V = \cup_i D_i$ consists of all such lattice nodes. The membership relation $R \subseteq X \times V$ specifies the lattice elements a data object "belongs to." We require that R be *upward closed*, that is, if $\langle x, \ell \rangle \in R$ and $\ell \leq \ell'$, then $\langle x, \ell' \rangle \in R$.

For $\ell \in V$, we define $X|_\ell$, read X *restricted to* ℓ, as $X|_\ell = \{x \in X \mid \langle x, \ell \rangle \in R\}$.

When there are several dimensions, we can endow them with a naturally defined lattice structure. For nonempty $D' \subseteq D$, the *multi-dimension* $MD(D')$ is defined as follows. If D' is a singleton, the multi-dimension is simply the sole dimension in D'. Otherwise, assume $D' = \{D_1, \ldots, D_d\}$. Then $MD(D')$ is again a lattice whose elements are $\{\langle \ell_1, \ldots, \ell_d \rangle \mid \ell_i \in D_i\}$, where $\langle \ell_1^1, \ldots, \ell_d^1 \rangle \vee \langle \ell_1^2, \ldots, \ell_d^2 \rangle = \langle \ell_1^1 \vee \ell_1^2, \ldots, \ell_d^1 \vee \ell_d^2 \rangle$, and likewise for $\wedge$. The membership relation R is then extended to contain $\langle x, \langle \ell_1, \ldots, \ell_d \rangle \rangle$ if and only if it contains $\langle x, \ell_i \rangle$ for all i.

3.1 Pairwise Disjoint Collections

Recall that a PDC is a set of restrictions of a multi-dimension that are conceptually disjoint. Formally, for any multi-dimension $MD(D')$ and any set $S = \{\ell_1, \ldots, \ell_d\}$ of elements of the multi-dimension, we say that S is a PDC if $\ell_i \wedge \ell_j = \bot$ for all i, j with $i \neq j$.

Two special classes of PDCs are *sequential* and *factored* PDCs, which we now discuss.

Intuitively, a sequential PDC divides data according to a single dimension, then recursively subdivides each part using a second dimension, and so on. A factored PDC is essentially a cross-product of PDCs in each dimension. Clearly, every factored PDC is a sequential PDC (with any ordering of the dimensions). For formal definitions, see [1].

A simple example that illustrates the difference between these classes of PDCs is the following, based on the MSDB of newspaper articles. The PDC {(Asia, 2003), (USA, 2003), (Asia, 2004), (USA, 2004)} is factored since it is the cross-product of the PDCs {Asia, USA} and {2003, 2004}. The PDC {(Asia, 2001–2002), (Asia, 2003), (USA, 2001), (USA, 2002–2003)} is a sequential PDC, divided first along the `geography` dimension and then along the `date` dimension. The PDC {(Asia, 2003), (USA, 2004)} is neither factored nor sequential.

There are two main reasons why sequential and factored PDCs arise naturally. First is the fact that these classes of PDCs may be constructed via algorithms that are considerably more efficient than general PDCs [1]. The second main reason to focus on sequential and factored PDCs is that they admit much more succinct representations, which could be conveyed to the user more compactly. All our implementations reported in this paper produce factored or sequential PDCs.

[1] A lattice is a set of elements closed under the associative, commutative binary operations meet ($\wedge$) and join ($\vee$) such that $a \wedge (a \vee b) = a \vee (a \wedge b) = a$ for all a and b; it is bounded if there are two elements $\top$ and $\bot$ such that $a \wedge \bot = \bot$ and $a \vee \top = \top$ for all a. A lattice also induces a partial order: $a \leq b$ iff $a \wedge b = a$.

3.2 Queries

One may construct a broad range of queries in the MSDB framework; as outlined in the Introduction, we will develop a principled approach for doing so. Namely, we will identify a broad class of optimization questions and point out the existence of fairly efficient polynomial-time algorithms for these. Later, in Section 4, we will present a significantly more efficient algorithm for the important special class of sum-additive functions.

The basic query structure we consider in this paper is the following:

Given: Dimensions $D' \subseteq D$, data objects $X' \subseteq X$, score function $f : 2^X \times D \to \mathbb{R}$, an associative binary combination operator $\circ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, and integer k. (Here $\mathbb{R}$ denotes the reals.)

Find: PDC $\{\ell_1, \ldots, \ell_k\} \subseteq MD(D')$ such that $g(X'; \ell_1, \ldots, \ell_k) = f(X'; \ell_1) \circ \cdots \circ f(X'; \ell_k)$ is maximized over all such PDCs.

We refer to a particular fixed pair $(f, \circ)$ as a *query type*. A simple example of $(f, \circ)$ is $f(A; \ell) = \#(A|_\ell)$ and $\circ = +$. The basic algorithms in [1], developed there for specific optimization problems, immediately yield efficient algorithms for single numerical and hierarchical dimensions for all query types $(f, \circ)$ that can be evaluated efficiently. Specifically, assuming unit cost for the evaluations of f and $\circ$, the algorithms in [1] can be easily modified to obtain an $O(n^2 k)$ time algorithm for a single numerical dimension and an $O(nk^2)$ time algorithm for single hierarchical dimension. While the case of hierarchical dimensions can be considered well-solved, the solution for numerical dimensions is too expensive, for example, when considering hourly sales data over large periods of time.

Note that, in general, an objective function $g(A; \ell_1, \ldots, \ell_k)$ need not admit a decomposition into f and $\circ$; an example is $g(A; \ell_1, \ldots, \ell_k) = \#(\cup_i A|_{\ell_i})$. Fortunately, a large class of optimization problems do admit such a decomposition, and this motivates the search for efficient algorithms for various query types $(f, \circ)$ depending on easily identifiable properties of the function f and the $\circ$ operation.

We say that a score function f is *monotone* if $f(A; \ell) \leq f(B; \ell)$ for all sets $A \subseteq B \subseteq X$ and every $\ell \in D$. Similarly, we say that a score function f is *additive* if two conditions are met:

1. There is a function $h : 2^X \to \mathbb{R}$ such that $f(A; \ell) = h(A|_\ell)$ for every $\ell \in D$.

2. $f(A \cup B) = f(A) + f(B)$ whenever $A \cap B = \emptyset$ and $A, B \subseteq X$.

An example of an additive score function is $f(A; \ell) = \#(A|_\ell)$. We say that $(f, \circ)$ is *min-monotone* if $\circ$ is min and f is monotone. Likewise, $(f, \circ)$ is *sum-additive* if $\circ$ is $+$ and f is additive.

Finding additional interesting classes of query types, as well as designing efficient algorithms for optimization problems not decomposable in this way, are exciting research questions with significant potential benefits.

4 Algorithms for sum-additive $(f, \circ)$ pairs over numerical dimensions

In this section we present a new algorithm for sum-additive query types over a single numerical dimension. Using this algorithm, we can compute sequential PDCs over any set of numerical dimensions in time linear in the number of input documents.

4.1 The maximum subinterval problem

Our algorithm is based on what we call the *k-maximum subinterval problem*. Informally, the problem is to find k non-overlapping intervals in a sequence of real numbers whose sum is maximized. More formally, given a sequence $x[1], \ldots, x[n]$ of real numbers and a positive integer k, find a set of at most k non-overlapping sub-intervals I_i such that $\sum_i S(I_i)$ is maximized, where, $S(I) = \sum_{\ell \in I} x[\ell]$.

First, we claim that the problem of finding optimal PDCs for sum-additive query types over numerical dimensions can be cast as the maximum subinterval problem. Given a subset X' of documents, a numerical dimension, and an integer k, consider the instance of the k-maximum subinterval problem with $x[i] = f([r_i, r_i])$, where $f(\emptyset) = 0$. Since f is additive, it is easy to see that solving the k-maximum subinterval problem is equivalent to finding the optimal PDC.

We now turn to an algorithm to solve the maximum subinterval problem. Without loss of generality, we can assume that the given input sequence $x[1], \ldots, x[n]$ is *alternating*, i.e., $\text{sign}(x[i]) \neq \text{sign}(x[i+1])$ for $i = 1, \ldots n-1$. This can be realized by simply replacing each consecutive sequence of positive (resp. negative) values by their sum. It is easy to see that the optimal solution is unchanged under this simplification. Similarly, we can assume that $x[i] \neq 0$ for all $i = 1, \ldots, n$.

For $k = 1$, there is a simple linear-time dynamic programming algorithm (folklore) for this problem. The algorithm is the following. Perform a single scan through the sequence $x[1], \ldots, x[n]$ from left to right, at each step maintaining the best interval seen so far. The scan is split into phases, as follows. Let i be the left end of the current phase. We begin scanning elements $x[j]$ where $j \geq i$, keeping track of $S([i, j])$. If $S([i, j])$ is negative, end the current phase and begin a new phase with $j+1$ as the left end. Otherwise, if $S([i, j])$ is greater than the current best interval, update the current best interval. At the end, output the interval with the maximum sum that was encountered. It is easy to see that this algorithm runs in linear time. To see its correctness, consider points j_1, the start of some phase; j_2, within that phase; and j_3, the last point of the phase, and assume that point $j_3 + 1$ exists. Then $S([j_1, j_2 - 1])$ is nonnegative, and $S([j_1, j_3])$ is negative, by construction; thus, $S([j_2, j_3])$ must be negative. If the optimal solution

begins at j_2 and ends in another phase, then it may be replaced by another solution beginning at j_3+1, removing $S([j_2,j_3])$ and hence improving its overall value, a contradiction. Thus, the optimal interval will not cross a phase boundary. By a similar argument, for any phase (including the last one), if the optimal solution begins at j_2 and ends within the same phase, its left endpoint may be extended to j_1, improving its value by $S([j_1, j_2-1])$, which is nonnegative. So there must exist an optimal interval that begins at a phase boundary, and does not cross a phase boundary; the algorithm considers all such intervals.

By negating all values in the sequence, the same algorithm may also be applied to find the subinterval with the minimum sum.

For general k, there is a simple quadratic-time dynamic programming algorithm to find the optimal solution. Let $P(i,k)$ be the optimal value produced by selecting k non-overlapping intervals of the sequence $x[1],\ldots,x[i]$. Let $B([j',i])$ be the value of the subinterval of $[j',i]$ with maximum sum. Note that $B([1,i])$ may be computed for all i by a single pass of the optimal $k=1$ algorithm above, so all B values may be computed in time $O(n^2)$. The optimal value for k intervals may then be computed by the following recurrence:

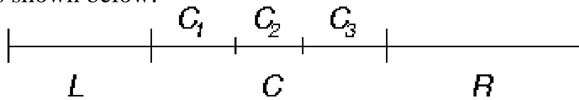
$$P(i,k) = \max_{j'<i}\{P(j'-1,k-1) + B([j',i])\},$$

It is easy to see that this algorithm takes $O(n^2k)$ time.

4.2 A new linear-time algorithm

In this section we present an $O(nk^2)$ algorithm for the k-maximum subinterval problem. From our discussion earlier, this implies a linear-time algorithm for sum-additive $(f,\circ)$ pairs over a single numerical dimension.

Let C be the subinterval of $[1,n]$ with the maximum sum, as determined by the $k=1$ algorithm above. Let L,R be the largest subintervals in $[1,n]$ to the left and right of C respectively. Let C_2 be the subinterval of C with the minimum sum, again as determined by the $k=1$ algorithm. Let C_1 and C_3 be the largest subintervals in C to the left and right of C_2 respectively. This results in a decomposition of the original interval into five subintervals, as shown below:



We show the following lemma, which will allow a recursive decomposition of the line. The proof is presented in Section 4.5.

Lemma 1. *There is an optimal solution such that*

1. *every interval of the optimal solution is either C, or lies entirely within some interval of the 5-way decomposition and*
2. *the optimal solution either contains C or contains an interval in both C_1 and C_3.*

Theorem 2. *The k-maximum subinterval problem can be solved in time $O(\min\{nk^2, \max\{nk, k5^k\}\})$.*

Proof. Given $x[1],\ldots,x[n]$, consider a 5-ary tree T with the following properties: first, each node of T corresponds to an interval; second, the root of T corresponds to the interval $[1,n]$; and third, the five children of a node correspond to the five intervals resulting from applying the 5-way decomposition defined above to the interval of the parent. Define the *heavy interval* of a node as the max-sum single interval that lies within the interval corresponding to the node, and define the *weight* $w(a)$ of the node a to be the value of the heavy interval. We show:

Lemma 3. *There is an optimal solution in which every interval is the heavy interval of some node of the tree T.*

Proof. Lemma 1 shows that there is an optimal solution that contains at least one interval in C; thus, L and R must have no more than $k-1$ intervals. The lemma further shows that in this constructed optimal solution, if there are two or more intervals within C, there must be at least one interval in each of C_1 and C_3. Therefore, there is a solution in which none of the five intervals of the decomposition contains more than $k-1$ intervals in this optimal solution.

Inductively, after j levels of decomposition, there is an optimal solution in which no interval in the 5-way decomposition contains more than $k-j$ intervals in the optimal solution. Therefore, after k levels of decomposition, there is an optimal solution in which each interval of the decomposition contains no more than one optimal interval.

Consider such an optimal solution and any single interval I of the solution. There must be some node of T that contains I and the interval corresponding to the node in the tree intersects no other interval of the optimal solution. Replace I in the optimal solution by the heavy interval of that node; this can only improve the solution. □

Thus, if we find the highest-score set of k non-overlapping heavy intervals from the tree, we will have found an optimal solution. This corresponds to finding k incomparable nodes of the tree (that is, k nodes such that no pair lie on a path from the root to any leaf) of maximum total weight. We can do this by dynamic programming, as follows. First, we modify the tree to make it binary. Each node has five children, so replace it with a depth-3 binary subtree. Assign one child each to five of the eight leaves of the subtree and weight $-\infty$ to all other nodes,[2] both leaf and internal, of the subtree. Any optimal solution of k incomparable nodes in the tree will never contain one of these new nodes, so a solution in this new tree can be mapped to a solution of equivalent cost in the original tree.

Let $B(a,k)$ be the best way to choose k nodes of the tree rooted at a, with children a_1,a_2, such that no two nodes lie on a root-leaf path. Then B may be computed by the following recurrence:

$$B(a,k) = \max\left\{w(a), \max_{k'=0}^{k}\left(B(a_1,k') + B(a_2,k-k')\right)\right\}.$$

[2]A sufficiently large real weight suffices.

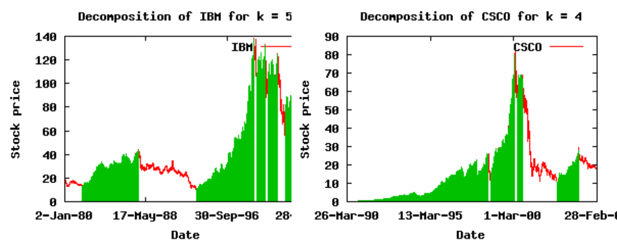


Figure 1: An illustration of k-maximum subintervals.

At each node in the tree, the time taken to compute $B(a,k)$ is $O(k)$. Since the number of nodes in the tree is at most $\min\{nk, 5^k\}$, the dynamic program runs in time $O(k\min\{nk, 5^k\})$.

It remains to consider the time to compute the tree. At each level of the tree, the intervals assigned to nodes at that level represent a partition of the original n points. Each interval of the partition must be processed in linear time, to compute the next-level decomposition, resulting in $O(n)$ time overall to process each level. Thus, the total time to compute the tree is $O(nk)$. So, the algorithm runs in time $O(\min\{nk^2, \max\{nk, k5^k\}\})$. □

4.3 Evaluation of algorithms

We implemented both the linear-time and quadratic-time algorithms. Both implementations are in Perl and no effort was made to optimize the code. The timings in Section 6 are performed on a machine with 256M of memory and a 950MHz P3 processor. Notice that the quadratic-time algorithm may be implemented in a straightforward manner using quadratic space to hold the best subinterval of each interval $[i,j]$; we adopted this implementation for simplicity. It is easy to reduce the space to linear if required. When the quadratic-time algorithm is run on five hundred documents, timings begin at ten seconds and grow to around one hundred seconds; timings on larger documents sets are prohibitive. On the other hand, for values of k up to five, the linear-time algorithm completes in under five seconds even for 100,000 data points, and in under one second for 10,000 data points.

4.4 An example application: Stock prices

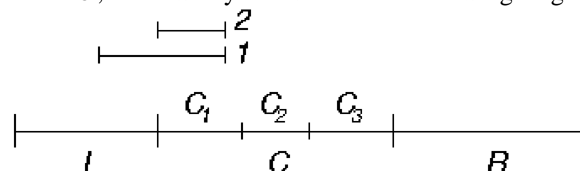
We illustrate an application of the sum-additive query types for single numerical dimension. Figure 1 shows the five (in the darkened-in) regions of greatest growth in IBM's stock price, and the four regions of greatest growth in Cisco's stock price, over the period from 1980 to 2005. The results are based on the difference between the stock price at the beginning of a day and the price at the beginning of the next day. The same formulation also applies to the log of the ratio of the prices, and then shows the regions of greatest aggregate growth.

4.5 Proof of Lemma 1

Consider some optimal solution Φ. We will show how to convert Φ into a new optimal solution that meets the conditions of the lemma. The proof will be by a series of modifications to Φ; we assume that the modifications are performed sequentially, so that the solution to which we apply modification i will already have been "processed" by modifications 1 through $i-1$. No modification will change the value of the solution.

MODIFICATION (A): Φ may be modified so that no interval crosses an endpoint of C.

Assume that Φ contains an interval that crosses an endpoint of C, as shown by interval 1 in the following diagram:



We modify Φ, replacing interval 1 with interval 2; this will not decrease the value of the solution. Otherwise interval $1 \setminus 2$ would be positive in value, and C could be extended and would become more positive, a contradiction to the maximality of C. Thus, our modified Φ contains no interval crossing an endpoint of C.

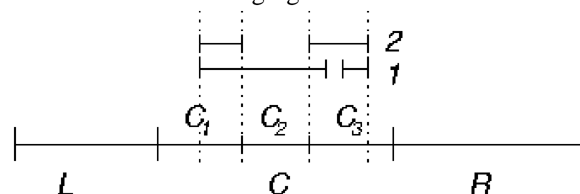
MODIFICATION (B): Φ may be modified to contain either interval C or two or more intervals within C.

Assume that Φ contains no interval in C. Then we replace any interval of Φ by C itself; as C has the highest score of any interval, this must not decrease the value of the solution. Thus, Φ contains no interval crossing an endpoint of C, and contains at least one interval in C. If only a single interval lies within C, then replace this interval with C itself; this will not decrease the value of the solution, and the conditions of the modification are met.

If C contains only a single interval then the conditions of the lemma are met, because this interval must be C itself, trivially meeting both conditions of the lemma. It remains to address the case in which C contains multiple intervals. Henceforth, we assume that the optimal solution under consideration contains multiple intervals within C.

MODIFICATION (C): If Φ contains an interval that covers C_2, it may be modified so that no interval covers C_2, and at least one interval appears in each of C_1 and C_3.

Consider the following figure.



Assume that an interval of Φ covers C_2, as in scheme 1 of the diagram. Since there are at least two intervals in C, there must be another interval to the left or right of the covering interval, as shown. Replace these two intervals with the two intervals shown in scheme 2 of the diagram.

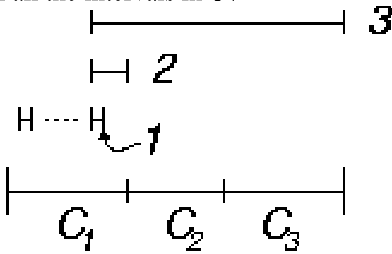
We show that this can only improve the solution. Notice that schemes 1 and 2 have the same left and right endpoints, and differ only in the "missing" interval in the middle. But scheme 2 is missing interval C_2, which is the interval of C with minimum sum, and so must have total sum at least as great as scheme 1.

MODIFICATION (D): Φ may be modified so that no optimal interval covers an endpoint of C_2.

If such an interval exists, modify Φ by removing the entries in C_2. The sum of these entries must be non-positive, or they could be removed from C_2, resulting in a new interval with smaller sum than C_2, a contradiction. Thus, the score of Φ will not decrease as a result of this modification.

We have now modified Φ so that no interval crosses an endpoint of C_2.

MODIFICATION (E): Φ may be modified so that C_1 does not contain all the intervals in C.



Consider interval 1 of the figure above, the rightmost optimal interval in C_1. This interval may be extended to the right until it includes the right endpoint of C_1, becoming interval 2; since C_2 is the most negative interval, the newly-covered region must be nonnegative, or C_2 could be improved.

Interval 2 may then be extended to interval 3, by adding C_2 and C_3. If this were to decrease the score of Φ then we would have $S(C_2) + S(C_3) < 0$. But in this case, C_1 must have a strictly greater score than C, a contradiction. We may now apply Modification (C) to replace interval 3, which covers C_2, with at least as strong a pair of intervals, one in each of C_1 and C_3.

The modified optimal solution Φ now meets all the conditions of the lemma.

5 System

Figure 2 shows a block diagram of a system for an MSDB over a large corpus of textual documents. There are three components in the system. The first, labeled "Document store/index backend," is a general backend capable of storing and indexing large collections of textual data. This system should be viewed as a black box that does the "heavy lifting" of large data handling and is able to provide in an efficient manner the random samples that the algorithms require to run. We describe the particular backend we employ in more detail below.

The box labeled "Multi-structural query engine" contains the implementation of all the algorithms described in this paper and in [1]. It generates a random sample from

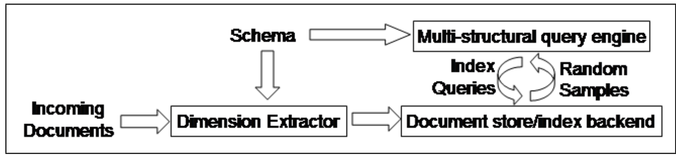


Figure 2: Block diagram of architecture for an MSDB over a large corpus of documents.

the document store/index backend, and then processes this random sample in order to generate a PDC.

Finally, the box labeled "Dimension extractor" processes documents as they arrive into the system, tagging them with information about where they belong in each dimension. In some cases, this box simply uses metadata that arrives with the document: some corpora arrive with a publication date or even a set of topic tags that may be converted directly into elements of R, the relation on documents and dimensions. Other types of metadata may be extracted by text processing, such as the entities mentioned in a document, or the topic of the document as determined by an automated classifier. A platform for document annotation such as UIMA [2] may be employed in this context.

The system works as follows. Documents are tagged as they arrive into the system. When a multi-structural query arrives, specifying a particular $X' \subseteq X$, $D' \subseteq D$, and $(f, \circ)$ pair, the query engine must process the query. It generates a random sample of X' of a specified size (discussed below), and also provides for each document x in the random sample the entries $R(x, \cdot)$ specifying where the document belongs in each dimension.

Sample sizes. As stated earlier, the backend produces a random sample of X' for processing by the multi-structural query engine. Clearly, with more random samples, the system will be able to return nodes of $MD(D')$ that represent smaller fractions of X'; however, more random samples means more computation time. For example, a random sample of size 1,000 means that elements of the multidimension will probably not be represented at all in the sample unless they contain roughly at least 1/1,000$^{\text{th}}$ of the documents of X'. In our experiments, we consider various sample sizes between 100 and 10,000.

Backend capabilities. The backend must provide the capability to produce the required random sample of X'. We describe our backend in detail later.

5.1 System description

5.1.1 Backend and dimensional extractor

We employ IBM's WebFountain system [3] to provide the backend and dimensional extractor modules. The flow through the system proceeds as follows. When a document first arrives in the system, it is processed by a series of mining agents that extract its entries and include them into the multi-structural relation R. For instance, the publication date of the document is extracted for the `date` dimension, the person entities appearing on the page are ex-

tracted for the `people` dimensions, and so on. The document is then annotated with tokens corresponding to the membership of the document in each dimension. For hierarchical dimensions, tokens are created for the upward closure of each node in the hierarchy, so if the document mentions entity Mel Gibson then this token will be created in addition to `movie stars` and `people`. The overhead of such a scheme is a blowup in the number of tokens corresponding to the average depth of the tree. If the index also keeps keyword tokens to allow constructions of relevant subsets $X' \subseteq X$, then the overall overhead of keeping tokens for nodes with hierarchical dimensions is vanishingly small compared to the overhead of maintaining a standard keyword index.

Numerical dimensions are indexed as is, but the indexer provides built-in support for range queries, so the document will be returned in response to queries for any containing interval.

The system then provides unbiased samples as follows. Each document is assigned a unique identifier computed as the hash of a primary key; for web content, this key is simply the URL of the document. The postings list for any token is then ordered by this hash value. Complex boolean queries are then computed by merging together postings, all of which are ordered according to the shared hash values. The prefix of an arbitrarily complex query is then a random subset of the entire result set.

5.1.2 Multi-structural query engine

The query engine is implemented as about 8,000 lines of Perl, including test code. It implements the basic $(f, \circ)$ dynamic programming algorithm for hierarchical and numerical dimensions, and provides a framework for computing sequential and factored PDCs for arbitrary collections of dimensions. It also implements more efficient algorithms for min-monotone PDCs, as described in [1], and for sum-additive PDCs, as described in Section 4. A simple optimizer selects the appropriate algorithm at each step. Finally, the system includes implementations of the six query types described in Section 5.3 below.

The query engine should be viewed as a reference implementation to compute multi-structural queries. It has not been optimized for performance. Our goal is to show that such a system can produce responses to complex multi-structural queries in times measured in seconds or tens of seconds, rather than hours; there are many future modifications that could provide further improvements and so the timing numbers should be viewed as upper bounds.

5.2 Data and Dimensions

The experiments were performed during February of 2005, based on a crawl that was current at that time, consisting of roughly four billion pages. The dimensions we considered are the following:

Dimension	Type	Size
`people`	Hierarchical	165M
`politicians`	Hierarchical	7.4M
`geography`	Hierarchical	177M
`media`	Hierarchical	96M
`star rating`	Numerical	62M
`europe`	Numerical	11M
`date`	Numerical	1.5B
`crawl date`	Numerical	3.8B

The `people` dimension includes subcategories such as `corporate leaders` and `popular figures`; the leaves of this dimension are names of specific people. A subtree of the `people` dimension is `politicians`, which we have used as a separate dimension. It includes groups such as `u.s. executives` and `heads of state`. The leaves of the `geography` tree are geographical locations throughout the globe. Similarly, `media` is a taxonomy of source types and includes a large set of newspapers and magazines organized by categories such as `college` and `u.s. top 100`. To calculate the `star rating` value, we counted the mentions of movie stars on each document. Similarly, the `europe` dimension is a count of mentions of European locations. Our `date` dimension is the date the page was created (for those documents where one could be confidently detected) and the `crawl date` is the date when the page was crawled.

5.3 Query types and queries

In this section, we present a family of six $(f, \circ)$ pairs. We describe each query type and give examples; we then perform measurements on a benchmark set of 28 queries drawn from these six query types. The first three query types described below appeared in [1] but have been recast into the $(f, \circ)$ framework here; the remainder are new to this paper. To specify a query, we must give X', D', and $(f, \circ)$. In all of our benchmark queries, the subset $X' \subseteq X$ must be specified according to a simple scheme: any restriction that uses the dimensions of the MSDB is allowable (for instance, all documents mentioning a location in Europe that were crawled in 2003), and these queries may be further restricted by requiring that a certain keyword also appear on the page, for instance, the keyword "tsunami." In all the queries we describe below, we will specify X' by giving a simple string such as "`geography`:Europe `people`:Mel Gibson `key`:Iraq," which should be read as all the documents that mention a European location, also mention Mel Gibson, and contain the keyword Iraq.

We now describe the individual $(f, \circ)$ pairs. In all cases, $\circ$ is either addition or min. The definition of f is typically more complex. We must describe how f evaluates a particular element of $MD(D')$ with respect to a particular X'.

Divide. The goal of this query type is to find a PDC that divides X' into roughly equal sized pieces, to generate a high-level understanding of where the mass of the data lies with respect to the selected multi-dimension. We describe

this query type in some detail, and then provide a more cursory walkthrough of the remaining types. In DIVIDE, we ask for a PDC $H = \{h_1, \ldots, h_k\}$ that is complete (i.e., contains all the documents of X'; see [1] for details) and that maximizes $f(h_1) \circ \cdots \circ f(h_k)$, where $f(X'; \ell)$ is the number of documents of X' that appear at element ℓ of the multi-dimension, that is,

$$f(X'; \ell) = \#(X'|_\ell),$$

and the combine function $\circ$ is min. Thus, a PDC $H = \{h_1, \ldots, h_k\}$ is given a score $f(h_1) \circ \cdots \circ f(h_k) = \min\{f(h_1), \ldots, f(h_k)\}$. The score of a PDC is therefore the size of the largest node, and the formulation finds the PDC that maximizes the smallest node, giving a balanced representation of the content of the documents. This is a min-monotone query type. A sample query of this query type is: partition the documents (X') that mention any movie star into k intervals of time (D') such that every document belongs to an interval, and the number of documents in each interval is roughly the same (i.e., every interval contains at least s documents for the largest possible s).

Differentiate. This query type returns nodes for which a larger relative fraction of the documents of X' appear than would be expected, given the statistics of some *background* set $B \subseteq X$. Thus, we define

$$f_B(X', \ell) = \frac{\#(X'|_\ell)}{\#(X')} - \frac{\#(B|_\ell)}{\#(B)}.$$

The combine function is addition. This query type is sum-additive. A sample query of this query type is: compare documents (X') that mention George Bush to documents (B) about politicians that do not mention George Bush, and return date ranges (D') in which the documents that mention Bush also mention other politicians.

Discover. This query type returns nodes of the multi-dimensions that are *distinct* with respect to a separate set $M \subseteq D$ of measurement dimensions. "Distinct" means that documents of X' located at ℓ are cohesive according to the metric implied by M, and that these documents are well-separated from the other documents of X', again according to the metric implied by M. These intuitions are formalized in [1]; for reasons of space we refer the reader to that paper, which defines:

$$f(X'; \ell) = \frac{\sum_{x \in X'|_\ell,\, y \in X' \setminus (X'|_\ell)} d_M(x, y)}{\#(X'|_\ell)\, \#(X' \setminus (X'|_\ell))} - \gamma \frac{\sum_{x, y \in X'|_\ell} d_M(x, y)}{\#(X'|_\ell)^2}$$

The combine function is sum. Here γ is a parameter of the query type that trades off the cohesion of the documents at ℓ against their separation from other documents. A sample query using this query type is: among all documents (X'), find categories (D') of people who tend to occur in documents (M) that mention Paris.

Growth. This query type finds regions of most rapid growth; Section 4.4 gives an example of finding regions of rapid growth of a stock. The query type is defined when D' is a single numerical dimension. The base elements of the lattice are assumed to be a partition of the numerical dimension into fixed-width intervals such as days or weeks, and all other intervals are assumed to be aligned with these base intervals. The growth $g(i)$ of base interval i, is defined as $\frac{1+\#(X'|i)}{1+\#(X'|_{i-1})}$. Then

$$f(X'; \ell) = \sum_{i \in \ell} \log(g(i))$$

and the combine function $\circ$ is addition. This query type is sum-additive. A query of this type is: among documents (X') that mention Mel Gibson, find date ranges (D') that show significant growth.

Recency. The goal of this query type is to find, for example, media types that have published more content on a particular topic during the last few weeks than their history would warrant. It scores nodes of ℓ according to the ratio of the density of documents appearing at the node during some recent interval RI compared to some earlier interval EI of some numerical dimension, say, `date`. Thus,

$$f(X'; \ell) = \frac{\#(X'|_{\ell, RI})}{\#(X'|_{\ell, EI})}$$

and the combine function is addition. This query type is sum-additive. A sample query of this type is: among all documents (X'), find restrictions (D') of geography and organizations in which coverage grew significantly over the last month.

Value. This query type applies a quality score to each individual page in a collection. Positive values mean that the page has positive quality and will overall add to the measure; negative values mean the opposite. The quality of a collection is the sum of the quality of the pages in the collection. We consider two quality scores. The first is *star rating*, which is the number of movie star entities mentioned on the page, minus some threshold value. The second is *Europe affinity*, which is the number of references to any European location on the page. If the score of a page x is given by $s(x)$ then:

$$f_s(X', \ell) = \sum_{x \in X'|_\ell} s(x)$$

and $\circ$ is addition. This query type is sum-additive. A sample query of this type is: among media documents (X'), find date ranges (D') that contain significant number of references to European locations (*Europe affinity*).

6 Experimental results

This section presents our experimental evaluation of the system and queries described above. To our knowledge,

MSDB queries represent optimization problems that have not been studied at the petabyte scale in previous work. Our goal in this section is not to compare different techniques for the computation of such queries, but rather to demonstrate that execution times measured in seconds or tens of seconds are attainable for a set of queries that are reasonably broad and representative over a data set that is large. An earlier paper [1] covered a series of multi-structural queries from a qualitative perspective, evaluating particular results to show utility. In this paper, our focus is rather on performance: all results in this section pertain to the feasibility of implementing such queries in a real system, and do not include user studies or other qualitative evaluation of the *nature* of the results.

6.1 Experimental environment

Our measurements are structured as follows. We break timing information into backend and frontend components. The backend represents the operation of generating a sufficiently large and accurate sample of the set $X' \subseteq X$, annotated with all relevant entries from the relation R for documents in the sample. Notice that relevant entries typically include all entries in the dimensions $D' \subseteq D$ given as part of the query, plus possibly some auxiliary information such as the date of the document for the recency query type. The frontend then processes the sample, computing f for elements of the multi-dimension as necessary, and applying the appropriate optimization algorithm to generate the resulting PDC.

Backend queries are processed by the WebFountain distributed index, described in [3]. Frontend queries are performed on a 1.2MHz IBM x335 server with 2GB of memory. The backend processing is performed by issuing network requests from this machines to the WebFountain index. Once results have been returned to the frontend, processing begins to compute the optimal PDC.

Backend processing is measured in two ways. First, we instrument the frontend to report the overall latency of generating the sample with all necessary metadata. Second, we instrument the index to measure the amount of time spent from receiving the query to generating the response. Thus, the first latency measurement includes marshaling and unmarshaling data at the client, network latency, and index latency; the second measurement covers only index latency. All backend requests are saved in a cache on the frontend server, and frontend-only timings are generated as end-to-end times given that the appropriate sample data is loaded from a local cache.

6.2 Backend timings

We consider "warm cache" and "cold cache" backend timings. The warm cache case might more accurately be referred to as "hot cache", since the query has been performed recently, and the results are assumed to be cached in memory on the index server, or in file system cache if the result requires processing of a significant amount of data.

Num	n	Index (ms)	Total (ms)
100	330	32.79	229.72
500	330	117.60	1035.62
1,000	330	189.71	1936.71
2,500	330	388.64	4475.30
5,000	330	792.09	8631.06
10,000	330	1413.48	15768.39

Table 1: Warm cache timings, averaged over 33 queries, for various sizes of document sets. The Num column indicates the number of documents with associated metadata retrieved per query. The n column gives the number of measurements of fetches of this size (10 per query). The Index column shows the number of milliseconds spent in the index to retrieve the relevant information. The Total column shows the total time spent gathering backend data, including marshaling and unmarshaling of content at the client, processing time in the index, and network latency.

Table 1 shows the aggregate statistics for warm cache backend timings over 28 benchmark queries. Of the 28 queries in our benchmark set, five are DIFFERENTIATE queries, each requiring a foreground and background sample. Thus, computing our entire benchmark requires 33 total distinct network requests. Queries performed with sample size 100 typically return very quickly, and represent a "high-level" result which may be sufficient to guide the user to more detailed queries. Samples of 10,000 documents require more overall time: 1.4 seconds in the index on average, with almost 16 seconds total latency. The significant difference in these timings is for two reasons. First, the amount of data and metadata being transferred is significant (we give details below). Second, the particular transport mechanism we employ is not optimized for very large frame sizes. Significant optimizations of these times is possible in a tuned system.

Table 1 shows average times only, so we now provide a more detailed view. Figure 3 shows a histogram of the time to complete a query for the same range of sample sizes of 1,000 documents and 10,000 documents. Backend speed almost never exceeds a second until we reach 10,000 document samples, and overall backend latency for 1,000 document samples, which is a reasonable tradeoff between detail and performance, is typically 1–4 seconds.

Cold cache timings are performed by streaming a large amount of data through the index between queries, in order to flush both the in-memory cache and the file system buffer cache. Our experiments showed that running a 100-second query was sufficient to perform this flush. Because each query requires 100 seconds of heavy processing, and the system is supporting customers, we were restricted to running tests at night and we performed a smaller number of trials. The aggregate results are shown in Table 2. Each query was run either once or twice for each sample of three sample sizes: 100, 1,000, and 5,000 documents. The index timings are higher than in the warm cache case, but the overall end-to-end latencies are not significantly larger.

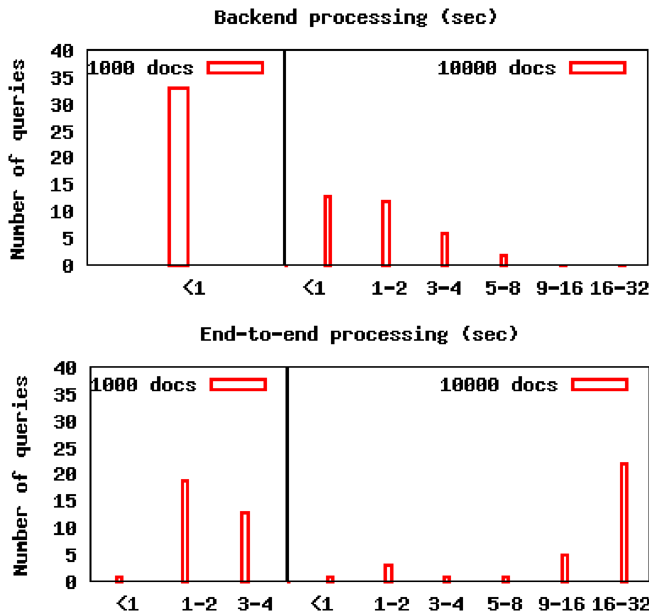


Figure 3: The top figure shows the histogram of the number of seconds spent in the backend generating all necessary data. The left frame shows 1,000 document samples and the right frame shows 10,000 document samples. The bottom figure shows the same results for end-to-end processing time.

Num	n	Index (ms)	Total (ms)
100	46	119.30	311.35
1,000	46	1292.07	3028.46
5,000	51	2099.35	9544.45

Table 2: Cold cache timings, averaged over 33 queries, for various sizes of document sets. The columns are to be interpreted as in Table 6.2.

We now explore the bytes of data and metadata being returned by these queries. Figure 4 shows a histogram of the cached size for sample sizes of 1,000 documents and 10,000 documents. Several queries at the 10,000 document size return in excess of 100M of data, which must be marshaled, transferred over the network, unmarshaled, and written to disk in the client cache.

6.3 Multi-structural query engine timings

We now turn to timings in the query engine. Table 3 shows the average time to compute a multi-structural query, broken by sample size and size k of the resulting PDC. The size of the resulting PDC is a much less significant contributor to the overall time than the sample size. Certain query types, of course, are quadratic in the sample size and hence show significant growth with sample size.

We now break out timing numbers by the six query types of Section 5.3, aggregated over 28 benchmark queries of a particular type. Figure 5 shows the results for PDCs of size $k = 5$ and $k = 10$, over various different sample sizes. For

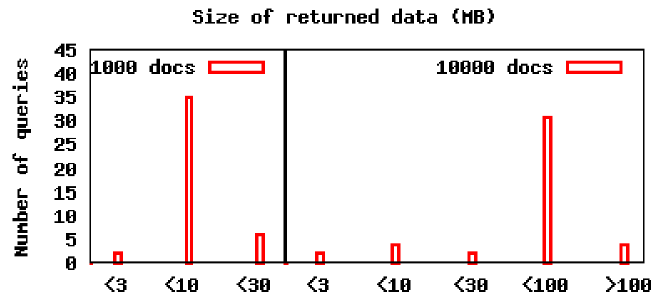


Figure 4: Sizes in megabytes for data returned per query. The left frame shows the results for 1,000 document samples and the right frame shows the results for 10,000 document samples.

Docs	$k = 5$	$k = 10$	$k = 15$	$k = 20$
100	1.0	1.1	1.2	1.3
1,000	9.2	9.6	9.6	10.0
5,000	43.0	44.2	45.7	48.2

Table 3: Average time in seconds to solve multi-structural query, after all data and metadata has been loaded from the backend, over 28 benchmark queries for various sample sizes and PDC sizes k.

both values of k, it is clear that certain query types have significantly higher processing times than others. These queries have more compute-intensive f functions, which dominate the runtime of the queries that take more than one minutes.

Figure 6 shows aggregate end-to-end timings of the entire system, from the network requests to generate the data and metadata, to the time to compute the optimal PDC in the query engine. For 100 documents, all queries complete in under five seconds, and most complete in under a second. For 1,000 documents, most queries complete in under 10 seconds, but some require as much as 40 seconds overall. For 5,000 documents, most queries complete in under a minute, and some require as much as three minutes. Higher query times are dominated by the query engine processing, suggesting that further algorithmic work and query engine tuning is an appropriate direction to improve performance.

References

[1] R. Fagin, R. Guha, R. Kumar, J. Novak, D. Sivakumar, and A. Tomkins. Multi-structural databases. In *Proc. 24th ACM Symposium on Principles of Database Systems*, 2005.

[2] D. Ferrucci and A. Lally. Building an example application with the Unstructured Information Management Architecture. *IBM Systems Journal, Special Issue on Unstructured Information Management*, 43(3):455–475, 2004.

[3] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to build a

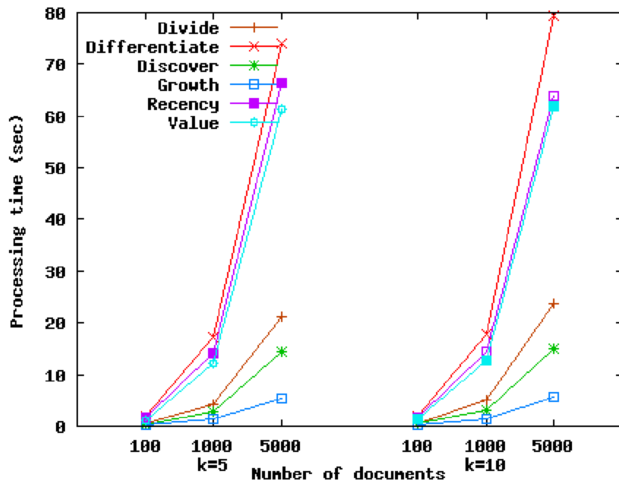


Figure 5: Average query engine timing over all queries within a query family for sample sizes of 100, 1,000, and 5,000 documents, after all backend activity has completed. Results for $k = 5$ are on the left, and for $k = 10$ on the right.

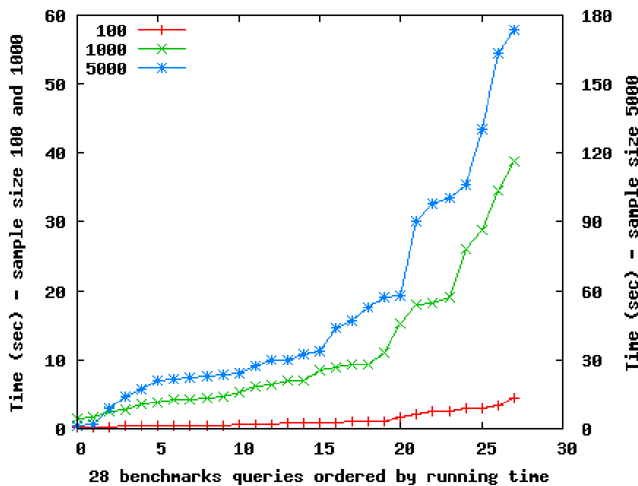


Figure 6: Overall system timings for 28 benchmark queries, ordered by running time, for sample sizes of 100, 1,000, and 5,000 documents, with $k = 5$. To make the scale of all three lines visible, we show times for 100 and 1,000 documents on the left axis, and for 5,000 documents on the right axis.

WebFountain: An architecture for very large-scale text analytics. *IBM Systems Journal*, 43(1):64–77, 2004.

[4] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 205–216, 1996.

[5] C. Hurtado. *Structurally heterogeneous OLAP dimensions*. PhD thesis, Computer Science Department, University of Toronto, 2002.

[6] H. V. Jagadish, L. S. Lakshmanan, and D. Srivastava. What can hierarchies do for data warehouses? In *Proc. 25th International Conference on Very Large Data Bases*, pages 530–541, 1999.

[7] M. Rafanelli, editor. *Multidimensional Databases: Problems and Solutions*. Idea Group, 2003.

[8] R. Torlone. Conceptual multidimensional models. In M. Rafanelli, editor, *Multidimensional Databases: Problems and Solutions*, pages 69–90. Idea Group, 2003.

[9] P. Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In *Proc. 10th International Conference on Scientific and Statistical Database Management*, pages 53–62, 1998.