Paper Number 259

Selecting and Maintaining Materialized Views for Message Management

Himanshu Gupta[*]
Stanford University
hgupta@db.stanford.edu

Divesh Srivastava[†]
AT&T Labs–Research
divesh@research.att.com

Abstract

Electronic messaging has become one of the primary means for the dissemination, exchange and sharing of information. This is facilitated, especially within an organization, by the use of shared folders, which are supported by current electronic messaging systems. We demonstrate that considerable additional flexibility can be achieved by modeling the messaging system as a data warehouse, where each message is a tuple of attribute-value pairs, and each folder is a view on the set of all messages in the messaging system; both user mailboxes and current-day shared folders can be treated as specialized views. Supporting this paradigm in emerging messaging systems, which support thousands of users, makes it imperative to efficiently support a very large number of folders, each defined as a selection view: this is the key difference with conventional data warehouses.

We identify two complementary problems concerning the design of such a messaging system. One of the most important tasks of the messaging system concerns the efficient incremental maintenance of eagerly maintained (materialized) folders. This problem for our model of folder definitions is a more general version of the classical point-location problem, and we design an I/O and CPU efficient algorithm for this problem, based on external segment trees and tries. A second important design decision that a messaging system needs to make is the choice of eagerly maintained folders. We present various special cases of the folder-selection problem in the context of messaging systems and present efficient exact/approximation algorithms, and complexity hardness results for them.

1 Introduction

Electronic messaging has become one of the primary means for the dissemination, exchange and sharing of information. This is facilitated in considerable part, especially within an organization, by the availability of shared folders, which allow message originators to share information on specific topics with, for example, their project members or department colleagues. The emergence of

[*]Contact author. Gates Computer Science Building, Wing 4B, Stanford CA 94305. Tel: (650) 723-9445, Fax: (650) 725-2588.

Supported by NSF grant IRI-96-31952.

[†]Member of the VLDB'98 Americas/Australia program committee.

electronic messaging systems that support several thousands of users in managing their information exchange, for example, Domino [Dom], Sun's Internet Mail Server [SIMS] and Oracle's Inter Office [OIO], and their rapid adoption by organizations, provides a common information repository of messages exchanged over time within an organization, that is akin to a data warehouse over message data.

Individual users' mailboxes and current-day shared folders can be regarded as simple materialized views in this data warehouse, selecting messages based on conditions satisfied by their "To" attributes. Richer forms of information sharing can be supported in this model by allowing folders to be defined as arbitrary views over the set of all messages in the messaging system, using values of the message attributes for defining the view. For example, one folder could contain all messages sent in February, 1998 to *any* member of the systems support group requesting for help with PCs; these messages could be shared amongst *all* PC experts in the systems support group. Another folder could contain all the message interactions between members of a project, including messages broadcast to the entire project team, and discussions between members of sub-teams of the project; this folder can serve as an archive of the evolution of the project. Note that for the definitions of these folders, the folder views have to be defined over the set of *all* messages in the messaging system and not just over the set of messages in a single user's mailbox. Clearly, the ability to automatically classify messages into folders based on conditions satisfied by multiple message attributes permits very flexible folder definitions.

Defining folders as views, and automatic classification of messages into folders also has the potential of helping individual users prioritize and effectively deal with the ever-increasing numbers of electronic mail messages received every day; manual classification, though effective when dealing with few messages, is too laborious a mechanism for adequately categorizing large numbers of messages. For example, one folder could contain messages from one's boss, another could contain all urgent messages that were received on a particular day, and yet another could contain calls for papers sent to the user as a member of the list dbworld@cs.wisc.edu. It is important to note that the folders need not partition the underlying messages, and a message could be contained in multiple folders. For example, an urgent message from one's boss would be contained in two of the above-mentioned folders.

In this paper, we identify and address two complementary problems that arise when folders are defined as views over the set of all messages in the messaging system data warehouse:

- Given a new message, which of the possibly large number of materialized folder views need to be updated to include this message?

  Since we expect an arbitrary message to be contained in only a few folders, while the number of folders supported by the messaging system may be very large, an algorithm that iteratively checks whether the new message is contained in each of the folders in the messaging system would be extremely inefficient. We devise an I/O and CPU efficient solution for the *folder-maintenance problem*, based on external segment trees [RS94] and tries [Fre60].

- Given a large set of folders defined as views, which of these folders should be eagerly maintained (materialized), and which of them should be lazily maintained, in order to conserve system resources while efficiently answering user queries?

We demonstrate that the general *folder-selection problem* is intractable, which motivates a study of special cases that arise in practice. We consider several alternative optimality criteria for many natural special cases, based on our model of messages and folders, and present efficient exact/approximation algorithms for them.

Both these problems arise in any data warehouse that supports materialized views. What distinguishes message management data warehouses from more conventional data warehouses are (a) the extremely large number of (folder) views defined in the (message management) data warehouse, and (b) the simple form of individual (folder) views as selections over the set of all messages in the data warehouse. While we focus on message management data warehouses in this paper, our techniques and solutions are more generally applicable to any data warehouse or multi-dimensional database with these characteristics.

The rest of this paper is organized as follows. In the next section, we briefly describe our model of the message management data warehouse. In Section 3, we consider the problem of efficiently maintaining folders, when new messages arrive into the messaging system. In Section 4, we discuss the problem of selecting an appropriate set of folders to eagerly maintain. We present related work for each of the two problems in their respective sections. We end with concluding remarks in Section 5.

2 The Message Management Data Warehouse

A data warehouse is a large repository of information available for querying and analysis [IK93, HGMW+95, Wid95]. It consists of a set of materialized views over information sources of interest, using which a family of (anticipated) user queries can be answered efficiently. In this section, we show that a message management system can be modeled as a data warehouse.

2.1 Messages and the Message Store

In the Internet community, an electronic mail message is considered to contain information of two types: *header fields*, which are used to convey control information; and a *body*, which is used to convey the actual data. The header fields of an electronic mail message are each identified by a keyword and a value (whose syntax may be dependent on the keyword). In contrast, the body is viewed as unstructured text. The syntax of an electronic mail message is defined in a document called RFC 822 [Cro82]. It should be noted, in particular, that the set of keywords in header fields of an electronic mail message is not fixed; the syntax permits header fields with new keywords to be easily added.

For the purpose of this paper, we model messages as having d attributes, $A_1, A_2, \ldots, A_d$: for header fields of the message, the keyword is the name of the attribute; the body of the message can be treated as the value of a new attribute called *X-Body*. The various examples in this paper use the commonly specified attributes *From*, *To*, *Date*, *Subject*, and *X-Priority* of electronic mail messages, with their obvious meanings. Each message m is represented as a tuple of d values $(b_1, b_2, \ldots, b_d)$, of which some may be unspecified.

The *message store* is the set of all messages in the system, and is the information source for the message management data warehouse.

2.2 Folder Views

A *folder* contains a set of electronic mail messages. Current day messaging systems support only folders that contain messages that have been manually classified as such by users. Here, we focus our attention on folders that are defined by views over the set of all messages stored in the underlying message store. The messages that are contained in the folders considered in this paper are determined automatically by the messaging system based on the folder definitions, and are *not* explicitly classified by the users. Manually populated folders can co-exist with automatically populated folders; the messaging system, however, does not concern itself with how the manually populated folders are maintained over time.

In this paper, we consider only folders that are defined as selection views on the message attributes as follows. The *atomic conditions* that are the basis of the folder definitions are of the forms "attribute *contains* value" and "attribute *arithop* value", where *arithop* $\in \{\leq, \geq, =, \neq, <, >\}$, and the values are from the domain of the attribute. Given a message attribute A_i, an *attribute selection condition* on A_i is an arbitrary boolean expression of atomic conditions on A_i. A *folder definition* is a conjunction of attribute selection conditions on the attributes of messages, i.e., we consider folders V defined using selection conditions of the form

$$\bigwedge_{i \in I} (f_i(A_i))$$

where $I \subseteq \{1, 2, \ldots, d\}$ is known as the *index set* of folder V and $f_i(A_i)$ is an attribute selection condition on attribute A_i. We expect the size of the index set $|I|$ of typical folders, that are defined as views, to be small (of the order of 1 to 5), compared to the number of message attributes (of the order of 25 to 30).

For example, the folder containing urgent messages received on December 31, 1999 may be defined as "($\wedge$ (*Date* = 31 Dec 1999) (*X-Priority* > 2))". A more sophisticated example, below, defines the folder containing messages written by Himanshu Gupta to Divesh Srivastava in 1997 as "($\wedge$ ($\wedge$ (*Date* $\geq$ 1 Jan 1997) (*Date* $\leq$ 31 Dec 1997)) (*To* = divesh@research.att.com) ($\vee$ (*From* = hgupta@db.stanford.edu) (*From* = hgupta@research.att.com)))". As an example of the use of text valued unstructured attributes, the folder containing calls for papers sent to the list dbworld@cs.wisc.edu may be defined as "($\wedge$ (*To* = dbworld@cs.wisc.edu) (*Subject contains* Call for Papers))".

It is important to note that our model of folders does not restrict folders to be defined only on the set of messages for an individual user, but allows folders to be defined on the set of *all* messages in the underlying message store. This approach has the advantage of permitting very flexible folder definitions, and clearly separates concerns of access control to the folders from the folder definitions. We do not discuss access control to personal and shared folders further, as it is outside the scope of this paper.

Each user in a messaging system defines some folders of interest. Any request to see all the messages in a specific folder is treated as a *folder query*; in our model, this is the only type of query permitted. The messaging system may decide to eagerly maintain some of the user defined folders; they are the materialized views in the message management data warehouse, and kept up to date, whenever a new message is stored in the message store. Other folders may be lazily maintained, and are computed whenever queried, using the messages in the message store and/or in the materialized folders.

2.3 Data Warehouse Design Decisions

Two of the most important decisions in designing a data warehouse are (a) the selection of materialized views to be stored at the warehouse, for a given family of (anticipated) user queries; such a selection is important given limited amount of resources such as storage space and/or total view maintenance time; and (b) the efficient incremental maintenance of the materialized views, for a given family of information source data updates.

In the following sections, we address these two key issues in the design of a message management data warehouse, taking into account the special characteristics that distinguish it from a conventional data warehouse (a) the extremely large number of folder views defined in the message management data warehouse, and (b) the simple form of individual folder views as selections over the set of all messages in the data warehouse. First, we deal with the problem of efficient maintenance of eagerly maintained (materialized) folders in response to new messages. In Section 4, we discuss the problem of determining which folders to materialize at the data warehouse.

3 The Folder-Maintenance Problem

In this section, we consider the folder-maintenance problem, i.e., the problem of efficiently updating a large set of materialized folders, when new messages are stored in the message store. We start with a precise definition of our problem, and present some related work, before describing our I/O and CPU efficient solution for the problem.

The problem we consider and its solutions are more generally applicable to any data warehouse that consists of a very large number of views defined using selections and unions over the underlying information sources.

3.1 The Definition

We formally define the *folder-maintenance problem* as follows. Let $V_1, V_2, \ldots, V_n$ be the large set of materialized folders in the messaging system. Given a new message $m = (b_1, b_2, \ldots, b_d)$, we wish to output the subset of folders that are affected by the arrival of m in the message store, i.e., the set of folders in which m needs to be inserted. We note here that a conventional message management system deals with a simple case of our general problem, where each user mailbox and shared folder is defined as a simple selection condition on the "To" field of the message.

Given the large number of folders in a message management system, the brute force method of sequentially checking each of the folder definitions to determine if the message needs to be inserted into the folder could be very expensive. The key challenge here is to devise a solution that takes advantage of the specific nature of our problem, wherein the size of the index set of typical folders is small, compared to the number of message attributes.

3.2 Related Work

The problem that we address here is a more general version of the classical *point-location problem*. In the point-location problem, the data set is a set of d-dimensional hyper-rectangles, the query is a point in the d-dimensional space, and the answer is the subset of all hyper-rectangles in the data set that contain the query point. An equivalent problem that has been examined by the database community is the predicate matching problem in active databases and forward chaining rule systems [HCKW90].

Our problem is more general than either the point-location problem or the predicate matching problem because each attribute selection condition defined using arithmetic operators may be a *union* of interval ranges (as opposed to just a single interval in the case of the point-location and predicate matching problems). This folder-maintenance problem can be reduced to the point-location (or predicate matching) problem by representing each folder view as a union of hyper-rectangles, but the number of such hyper-rectangles could be much larger than the original number of folder views.

There has been a considerable amount of work on the point-location problem from a theoretical perspective [EM81, Ede83a, Ede83b, Cha83]. Most of the effort there has been to design algorithms with good worst-case *main-memory* computation time complexity. The best known worst-case bound for a static data structure (i.e., no updates) is $\log^{d-1} n$ time to answer a query, with $n \log^{d-2} n$ storage space [Cha83]. The best worst-case bound known for the dynamic data structure is $\log^d n$ time to answer a query or to update the data structure, with the storage space requirement being $n \log^d n$ [EM81]. These algorithms are practically feasible only for very small values of d. Also, these algorithms do not have any better counterparts for secondary memory accesses.

The practical need for good I/O support has led to the development of a large number of external data structures, which do not have good theoretical worst-case bounds, but have good average-case behavior for common spatial database problems. Examples are the various R-trees, cell-trees, hB-trees. (Due to space limitations we refer the reader to [Sam89a, Sam89b] for a survey and applications.)

There hasn't been any work reported on designing secondary memory algorithms for the point-location problem, that have optimal worst-case bounds. However, there has been some recent work [KRVV93, RS95, VV96] reported for the dual problem of range-searching (the data set is a set of d-dimensional points, the query is a d-dimensional hyper-rectangle, and the answer is the subset of all points in the data set contained in the query hyper-rectangle) in two or three dimensions; the algorithms presented have optimal worst case bounds in the number of secondary memory accesses.

It is important to note that *none* of the previously proposed algorithms take advantage of small index sets of folder views, i.e., all of them treat unspecified selection conditions as intervals covering the entire dimension.

We start with a simple approach below and then, present our approach which is very efficient both in terms of the number of secondary memory (disk) accesses, and in terms of the CPU utilization.

3.3 Grouping "Similar" Views

In this subsection, we outline a simple approach for updating eagerly maintained folders, whose definition restricts each message attribute to be in a single interval, i.e., each folder is a hyper-rectangle. The approach is based on the hypothesis that a typical folder view will have a very small index set, compared to the number of message attributes.

Consider n folders $V_1, V_2, \ldots, V_n$, where each folder V_i is defined as[1] $V_i = \bigwedge_{j \in I_i} (A_j \in c_{ij})$, and c_{ij} is an interval on the domain D_j of attribute A_j. First, we partition the views into groups according to their index sets, i.e., views with the *same index set* get mapped to the same group. Then, we use the classical computational geometry approaches [Cha83] to solve the point-location problem in each group independently.

The time complexity of this approach depends upon the sizes of the groups. If the groups are $s_1, s_2, \ldots, s_k$, then the query time (in number of disk accesses) is given by $\sum_{i=1}^{k} \log_2^{l_i - 1}(|s_i|)$, where l_i is the size of the index set of s_i. The total space requirement for this method is $\sum_{i=1}^{k} |s_i| \log_2^{l_i - 2}(|s_i|)$. For a dynamic data structure, the query or update time is given by $\sum_{i=1}^{k} \log_2^{l_i}(|s_i|)$, with the total space requirement being $\sum_{i=1}^{k} |s_i| \log_2^{l_i}(|s_i|)$. When all groups have small index sets, as in the case of folder definitions, it is easy to see that this approach can be considerably superior to using the classical computational geometry approaches directly in d-dimensions.

The problem with this approach is that it doesn't generalize to more general attribute selection conditions.

3.4 Independent Search Trees Algorithm

In this subsection, we present our I/O and CPU efficient approach called the *Independent Search Trees Algorithm* for solving the folder-maintenance problem. For ease of understanding, we start with a description of the algorithm for folder definitions where each message attribute is restricted to be in a single interval, i.e., each folder is a hyper-rectangle. Later, we will extend it to general boolean expressions in the attribute selection conditions.

[1] We adopt this notation for simplicity. More precisely, each attribute selection condition would have to be a conjunction of two atomic conditions.

Folder-Maintenance of Hyper-Rectangles: Consider n folders $V_1, V_2, \ldots, V_n$, where each folder V_i is defined as

$$V_i = \bigwedge_{j \in I_i} (f_{ij})$$

and each f_{ij} is of the form $(A_j \in c_{ij})$ for some interval c_{ij} on the respective domain D_j. The data structure we use consists of d external segment tree structures [RS94] $T_1, T_2, \ldots, T_d$, such that tree T_j stores the intervals $\{c_{ij} \mid j \in I_i, 1 \leq i \leq n\}$.

We compute the set of affected folders (views) as follows. We keep an array I of size n, where $I[i] = |I_i|$, the size of the index set of folder V_i, initially. When a message $m = (b_1, b_2, \ldots, b_d)$ arrives, we search for intervals in the external segment trees T_j that contain b_j, for all $1 \leq j \leq d$. While searching in the segment tree T_j for b_j, when an interval c_{ij} gets hit (which happens when $b_j \in c_{ij}$), the entry $I[i]$ is decremented by 1. If an entry $I[i]$ drops to zero, then the corresponding folder V_i is reported as one of the affected folders. These are precisely the folders in which the message m will have to be inserted.

Handling Unstructured Text Valued Attributes: We now extend our Independent Search Trees Algorithm to handle unstructured text valued attributes. Presence of unstructured text attributes means that the folder definitions may now use the *contains* operator, i.e., f_{ij} can be $(A_j$ *contains* $s_{ij})$ for some string s_{ij} and a text valued attribute A_j. To incorporate the *contains* operator in our folder definitions, we build *trie* [Fre60] data structures instead of segment trees for text valued attributes. The question we wish to answer is the following. Given a set of strings $S_j = \{s_{ij} \mid j \in I_i\}$ (from the folder definitions) and a query string b_j (the value of the message attribute A_j), output the set of strings $s_{i_1 j}, s_{i_2 j}, \ldots, s_{i_l j}$ such that b_j *contains* $s_{i_p j}$ for all $p \leq l$. The *trie* data structure can be easily modified to answer the above problem. We build a *trie* on the set S_j of data strings. On a query b_j, we search the *trie* data structure for superstring matches for each suffix of b_j. This can be achieved in $(|b_j|^2 + l)$ character comparisons, where $|b_j|$ is the size of the query string b_j and l is the number of strings reported. The space requirements of the *trie* data structure is $k|\Sigma|$ characters for storing k strings, where $|\Sigma|$ is the size of the alphabet. Note that the *trie* yields itself to an efficient secondary memory implementation, as it is just a special form of a B-tree.

Folder-Maintenance of Boolean Expressions: When f_{ij}, the attribute selection condition involving attribute A_j, is $A_j \notin d_{ij}$ for some interval d_{ij} on domain D_j, we still store d_{ij} in the segment tree T_j. But, whenever d_{ij} is hit (which happens when $b_j \in d_{ij}$), we increase I_i to d (instead of decrementing by one), guaranteeing that folder V_i is not output. Similarly, we handle f_{ij}'s of the form $\neg(A_j$ *contains* $s_{ij})$ for an unstructured text valued attribute A_j. Also, in an entry $I[i]$ of array I, we store the size of *positive index set* of V_i instead of the size of V_i's index set. The positive index set I_i^+ of a view V_i is defined as $\{j \mid (j \in I_i) \wedge (f_{ij}$ is either of the form $(A_j \in c_{ij})$ or $(A_j$ *contains* $s_{ij}))\}$. Similarly, the *negative index set* I_i^- is defined as $\{j \mid (j \in I_i) \wedge (f_{ij}$ is either of the form $(A_j \notin d_{ij})$ or $\neg(A_j$ *contains* $s_{ij}))\}$.

The generalization to arbitrary boolean expressions for arithmetic attributes is achieved as follows. An arbitrary boolean expression f_{ij} for an arithmetic attribute A_j can be represented as $\bigvee_k(A_j \in c_{ijk})$ or as $\bigwedge_k(A_j \notin d_{ijk})$, for some set of intervals c_{ijk} or d_{ijk} on D_j. A segment tree T_j, corresponding to the attribute A_j, is constructed as including all the intervals c_{ijk} or d_{ijk} and corresponding entries in I are decreased by 1, or increased to d on hits to intervals appropriately. If A_j is an unstructured text valued attribute, then the only kind of boolean expressions that we can handle easily are of the type $f_{ij} = (\bigvee_k(A_j \textit{ contains } s_{ijk}))$ or of the type $f_{ij} = (\bigwedge_k \neg(A_j \textit{ contains } s_{ijk}))$ for a set of strings s_{ijk}. For such general cases of boolean expressions, the definitions of I_i^+ and I_i^- are appropriately extended to $\{j \mid (j \in I_i) \wedge (f_{ij}$ is either of the form $\bigvee_k(A_j \in c_{ijk})$ or $\bigvee_k(A_j \textit{ contains } s_{ijk}))\}$ and $\{j \mid (j \in I_i) \wedge (f_{ij}$ is either of the form $\bigwedge_k(A_j \notin d_{ijk})$ or $\bigwedge_k \neg(A_j \textit{ contains } s_{ijk}))\}$ respectively.[2]

Handling Unspecified Message Attributes: A message m is inserted into a view V_i based on the values of only those message attributes that belong to V_i's index set I_i. However, an attribute selection condition f_{ij} can be defined to accept or reject an unspecified attribute value. For example, it is reasonable for the selection condition (*X-Priority* > 2) to reject messages that have the attribute *X-Priority* unspecified, while an unspecified attribute value should probably pass the selection condition *X-Priority* $\neq 2$.

To handle such cases of unspecified attribute values in arriving messages, we maintain two disjoint integer sets P_j and F_j for each attribute A_j, in addition to its segment tree T_j. The *pass* list P_j is defined as $\{i \mid (j \in I_i^+) \wedge (f_{ij}$ accepts unspecified values$)\}$. Similarly, the *fail* list F_j is defined as $\{i \mid (j \in I_i^-) \wedge (f_{ij}$ rejects unspecified values$)\}$. Thus, if an arriving message m has its A_j attribute's value unspecified, we decrement the entry $I[i]$ by one for each $i \in P_j$ and increment $I[i]$ to d for each $i \in F_j$, instead of searching in the segment tree T_j.

CPU Efficient Initialization of Array I: Whenever a new message arrives in the messaging system, we need to initialize *each* entry $I[i]$ of the array I to $|I_i^+|$, the size of the positive index set of V_i. A simple scheme is to explicitly store (in persistent memory) $|I_i^+|$ for each V_i and initialize each of the n entries of I every time a new message arrives. However, since the message is expected to affect only a few folders, initializing all n elements of the array I can be very inefficient. Below, we present an efficient scheme to find the initial values in I, only for potentially relevant folders.

We number the views in such a way that $|I_j^+| \leq |I_k^+|$ for all $1 \leq j < k \leq n$. We define $d-1$ numbers $t_1, t_2, \ldots, t_{d-1}$, where t_j is such that $|I_{t_j}^+| < |I_{t_j+1}^+|$, i.e., these $d-1$ numbers define the transition points for the initial values in the array I. If no such t_j exists for some $j < d$, then $t_l = n+1$ for all $j \leq l < d$. We create a persistent memory array T of size d, where $T[0] = 0$ and $T[i] = t_i$ for $1 \leq i < d$.

On a hit to an interval c_{ijk}, we need to find the current value of $I[i]$ and decrement it by 1. Given i, we find the initial value of $I[i]$, $|I_i^+|$, as follows. We find a number x such that $T[x-1] < i \leq T[x]$

[2]We assume that $|I_i^+| > 0$ for each i. Else, we will need to keep a list of views with zero $|I_i^+|$ and report them if the entry $I[i] = |I_i^+|$ remains unchanged.

in $O(\log d)$ main-memory time. It is not difficult to see that $|I_i^+| = x$. On the first hit to an interval from folder V_i, we initialize $I[i]$ to $x-1$, else we reduce the *current* value of $I[i]$ by 1. How do we find out if V_i has been hit before or not? We do this by keeping a bit vector H of size n, where $H[i] = 1$ iff V_i has been hit before. Both arrays H and I can reside in main-memory, even when a few million folders are maintained as materialized views. For each new message that arrives into the messaging system, only the bit vector H needs to be reset to 0, which can be done very efficiently in most systems. Note that since the array T contains only d elements, it is quite small and hence can be maintained in main memory.

To reduce the number of disk accesses further, whenever an entry $I[i]$ is incremented to d, the corresponding entry of H, $H[i]$, is changed to 2 (which also makes each entry of H of size two bits). Now, whenever an interval c_{ijk} is hit, the entry $I[i]$ is initialized to $|I_i^+| - 1$ ($|I_i^+|$ is looked up using T) if $H[i] = 0$, the *current* value of $I[i]$ is decremented if $H[i] = 1$, and the hit is ignored if $H[i] = 2$. On a hit to an interval d_{ijk}, the entry $I[i]$ is changed to d, unless $H[i] = 2$.

Algorithm 1 **Independent Search Trees Algorithm**[3]

Given: Views $V_1, V_2, \ldots, V_n$ defined over the message store M, where each tuple/message in M has d attributes $A_1, A_2, \ldots, A_d$, and $m = \{b_1, b_2, \ldots, b_d\}$, the new message.

Let $V_i = \wedge_{j \in I_i}(f_{ij})$, where $I_i \subseteq \{1, \ldots, d\}$ and f_{ij} is $\bigvee_k (A_j \in c_{ijk})$ or $\bigwedge_k (A_j \in c_{ijk})$

Let $T_1, T_2, \ldots, T_d$ be the external-memory segment trees, where T_j contains all defined c_{ijk} or d_{ijk}s.

Also, let P_j and F_j be pass and fail lists, as defined above, for all $1 \leq j \leq d$

BEGIN

Reset $H[1..d]$ to zeros.

for $j = 1$ to d

if b_j is unspecified **then**

for each $i \in P_j$

Decrement(i);

endfor

for each $i \in F_j$

$I[i] = d$;

$H[i] = 2$;

endfor

else

Search b_j in T_j

for each c_{ijk} interval hit in T_j, i.e., for each c_{ijk} such that $b_j \in c_{ijk}$

Decrement(i);

endfor

for each d_{ijk} interval hit in T_j, i.e., for each d_{ijk} such that $b_j \in d_{ijk}$

$I[i] = d$;

$H[i] = 2$;

[2] For simplicity and better understanding, we ignore the extension to text valued attributes in this algorithm.

endfor
endif
endfor
END.
function Decrement(i):
Begin
if $H[i] = 0$ **then**
$H[i] = 1$;
$I[i] = |I_i| - 1$;
elseif (H[i] = 1) **then**
$I[i] = I[i] - 1$;
endif
if $(I[i] = 0)$ **then**
REPORT V_i;
endif
End ◇

I/O Efficiency: We now analyze the time taken by the above algorithm to output the folders affected, when a new message arrives in the messaging system.

Let B be the I/O block size. We define K_j as the number of intervals in the various folder definitions involving attribute A_j (equivalently, the number of entries in the segment tree T_j). Also, let m_j be the maximum number of intervals in tree T_j that overlap. Using the optimal external segment tree structure of [RS94], we can perform a search in a segment tree T in $\log_B(p) + 2(t/B)$ number of I/O accesses, where p is the number of entries in T and t is the number of intervals output. Therefore, the overall query time complexity of the above algorithm is $O(\sum_{j=1}^{d} \log_B(K_j) + 2(\sum_{j=1}^{d} m_j)/B)$.

For the purposes of computing number of I/O accesses, an unspecified value in attribute A_j of the new message m is treated just like a specified value, except that T_j is not searched for computing the hits, and $m_j = (|P_j| + |F_j|)$.

Theorem 1 *Consider n folder views whose definitions are conjunctions of arithmetic attribute selection conditions, where each attribute selection condition on attribute A_i is a boolean expression of atomic conditions on A_i.*

The above described Independent Search Trees Algorithm for folder-maintenance has a maintenance time of $O(\sum_{j=1}^{d} \log_B(K_j) + 2(\sum_{j=1}^{d} m_j)/B)$ disk accesses, where K_j and m_j are defined as above.[4] *The update time of the data structure due to an insertion of a new view is $O(\sum_{j=1}^{d} \log_B(K_j))$ disk accesses.*[5]

[4]If the array I can't be accommodated in main memory, then the number of accesses increases by $\sum_{j=1}^{d} m_j$.

[5]The array T can be maintained periodically.

For an unstructured text attribute A_s, where an attribute selection condition uses the contains *operator, the maintenance time required to handle A_s is $(|b_j|^2 + m_j)/B$, where $|b_j|$ is the length of the A_s attribute string b_j in the arriving message.* □

One of main features of the above described algorithm is that the time-complexity directly depends upon the total number of attribute atomic selection conditions specified, rather than n, as with all previously proposed algorithms for similar problems.

We experimentally compared the two approaches from Sections 3.3 and 3.4 for random instances of simple views that had atomic conditions as selection conditions. We observed that for various probability distributions across attributes, the second approach uses far fewer disk accesses, providing some preliminary experimental evidence of the I/O efficiency of the second approach above.

4 The Folder-Selection Problem

An important decision in the design of a messaging system is to select an appropriate set of folders to be eagerly maintained (materialized). The rest of the folders are lazily maintained, and are computed whenever queried, using the materialized folders and/or the message store. A natural optimization criterion is to minimize the storage space and/or folder maintenance time, while guaranteeing that each folder query (the request to see all messages in a specific folder) can be answered within some threshold of the optimal time. As the eagerly maintained folders can be regarded as materialized views in a data warehouse, the various folder-selection problems identified in this section can be looked upon as view-selection problems in a data warehouse.

Note that the notion of eagerly/lazily-maintained folders is different than the notion of important/unimportant folders. The issue is to select the most "beneficial" folders to materialize, so that *all* user defined folders can be answered within their respective query-time thresholds. The importance of a folder is implicitly specified by its query threshold which is essentially the query-time it can tolerate. In order to answer all the user defined folders within their thresholds, only a subset of the full set of folders may be materialized. Materializing a small subset of folders saves storage space and maintenance time, i.e., the time required to keep the eagerly maintained folders up to date in response to newly arriving messages. Below, we look at two possible efficiency considerations: bounding the additional query time for each folder, or bounding the average additional query time over all folders.

In this section, we first formulate the general problem of selecting folders to be eagerly maintained (materialized), and show that it is, unfortunately, intractable. We then take advantage of our model of folder definitions as selection views, and present some efficient exact/approximation algorithms, and complexity hardness results for the problems.

As with the previous problem of folder-maintenance, the problems addressed here are more generally applicable to the selection of views to materialize in a data warehouse, when the queries are restricted to selections and unions over the underlying sources of information.

4.1 General Problem of Folder-Selection

Consider a labeled bipartite hypergraph $G = (Q \cup V, E)$, where Q is a set of user-specified folder queries and V is a set of candidate folders (views) to be materialized. We refer to this graph as a *query-view* graph. This notion of a query-view graph is similar to that used in [GHRU97], but more general. E is the set of hyperedges, where each hyperedge is of the form $(q, \{v_1, v_2, \ldots, v_l\})$, $q \in Q$, and $v_1, v_2, \ldots, v_l \in V$. Each hyperedge is labeled with a *query-cost* of t, signifying that query q can be answered using the set of views $\{v_1, v_2, \ldots, v_l\}$ incurring a cost of t units. With each query node $q \in Q$, there is a query-cost threshold T_q, and with each view node $v \in V$, there is a weight (space cost) $W(v)$ associated.

We define the *folder-selection* problem as follows.

Folder-Selection Problem: *Given a bipartite query-view hypergraph G defined as above, select a minimum weighted set of views $M \subseteq V$ to materialize such that for each query $q \in Q$ there exists a hyperedge $(q, \{v_1, v_2, \ldots, v_l\})$ in G, where views $v_1, v_2, \ldots, v_l \in M$ and the query-cost associated with the hyperedge is less than T_q.*

The above problem is trivially in `NP`. Also, as there is a straightforward reduction (see Section 4.2) from `minimum set cover` to a special case of the folder-selection problem when G has only simple edges, the folder-selection problem is also `NP`-hard. The above folder-selection problem is exactly the problem of minimizing the number of leaves scheduled in a 3-level AND/OR scheduling problem with internal-tree precedence constraints [GM97]. The 2-level version of the AND/OR scheduling problem with internal-tree precedence constraints is equivalent to `minimum set cover`, while the 4-level AND/OR scheduling problem with internal-tree constraints is as hard as the `LABEL-COVER` [ABSS93, GM97] problem making it quasi-`NP`-hard[6] to approximate within a factor of $2^{\log^{1-\gamma} n}$ for any $\gamma > 0$. To the best of our knowledge, nothing is known about the 3-level version of the AND/OR scheduling problem with internal tree constraints.

The intractability of the general problem leads us to look at some natural special cases that arise in practice in the context of message management, and we present efficient algorithms for each of them. Recall that, in Section 2, we allowed folders to be defined only as selection views of a specified form. In this case, a folder needs only the union ($\cup$) and selection (σ) relational operators to be computed from a set of other folders. However, the above formulation of the folder-selection problem is much more general.

In the next subsection, we restrict the computation of a folder from other folders to just using the selection operator. We handle the case of using both the union and the selection operators in the subsequent subsection.

[6]That is, this would imply `NP` $\subseteq$ `DTIME`$(n^{poly(logn)})$. "A proof of quasi-`NP`-hardness is good evidence that the problem has no polynomial-time algorithm" [AL95].

4.2 Queries as Selections over Views

In this subsection, we consider a special case of the general folder-selection problem. We select a set of eagerly maintained folders (views) that will allow any other folder to be computed from the eagerly maintained folders using only the relational selection (σ) operator. As a query uses exactly one view for its computation, this implies that the query-view graph defined earlier will have only simple edges.

The folder-selection problem with the above restriction has a natural reduction from the `minimum set cover` problem and hence is also `NP`-complete. However, there exists a polynomial-time greedy algorithm that delivers a competitive solution that is within $O(\log n)$ factor of an optimal solution, where n is the number of folder queries, as shown below.

Algorithm 2
Given: A query-view bi-partite graph $G = (Q \cup V, E)$ with only simple edges.
BEGIN
  Remove all edges $(q, v) \in E(G)$, where the cost c associated with
    (q, v) is greater than T_q.
  $M = \phi$;
  REPEAT
    Select a view $\overline{v} \in V$ that has the most number of edges incident.
    $M = M \cup \{\overline{v}\}$;
    for each $(q, \overline{v}) \in E(G)$
      $Q = Q - \{q\}$;
      Remove the vertex q and all the incident edges on q from G.
    endfor
  UNTIL $Q = \phi$;
END $\diamond$

The above greedy algorithm is almost the same as that used to approximate the `weighted set cover` problem [Chv79]. It can be shown that the solution delivered by the above algorithm is within $O(\log n)$ of the optimal solution.

So far, we have not taken any advantage of the specific nature of the atomic conditions used in the folder definitions. We now do so, and restrict ourselves to folders defined using arithmetic operators on the message attributes. As the arithmetic operators used in an atomic condition assume an order on the domain, all defined folders form some sort of "orthogonal objects" in the multidimensional space of the message attributes. We take advantage of this fact and formulate a series of problems, presenting exact or approximate algorithms.

4.2.1 Folder-Selection with Ordered Domains

Consider an ordered domain D. Consider folders (views or queries) that are ranges over D. In other words, views and queries can be represented as intervals over D. As we restrict our attention to using only the selection operator for computing a query, a query interval q can be computed using a view interval v only if v completely covers q. With each pair (q, v), where v completely covers q, there is a query-cost associated, which is the cost incurred in computing q from v.

We observe here that the techniques used to solve the various problems of one-dimensional queries/views addressed in this section can also be applied to the more general case when the queries involve intervals (ranges) along one dimension and equality selections over other dimensions.

Problem (One-dimensional Selection Queries) *Given a set of interval views V and interval queries Q over an ordered domain D, select a minimum weighted set of interval views M such that each query $q \in Q$ has a view $v \in M$ that completely contains q and answers the query q within its query-cost threshold T_q.*

There is an exact dynamic programming algorithm as shown below that delivers an optimal solution to the above problem. The time complexity of the algorithm is $O(n^2)$, where n is the number of query and view intervals.

Algorithm 3
Given: Sets of intervals, Q and V, over an ordered domain D.
BEGIN
Let the array A_Q contain the set of intervals Q sorted by their rightmost points
for $i = 1$ to $|Q|$
$C[i] = \textit{infinity}$;
for each $v \in V$ that intersects $A_Q[i]$
Let j be the smallest integer less than i such that $A_Q[j]$ intersects with v.
if no such j exists **then**
$S[i] = < v >$;
if $(Cost(v) + C[j]) < C[i]$ **then**
$C[i] = Cost(v) + C[j]$;
$S[i] = concat(v, S[j])$;
endif
endfor
endfor
RETURN $S[|Q|]$;
END $\diamond$

The restricted version of the folder-selection problem considered here is similar to the view-selection problem in OR view graphs defined in [Gup97] with different optimization criteria and constraints. Gupta [Gup97] presents a simple greedy approach to deliver a solution that is within a constant factor of an optimal solution. We take advantage of the restricted model of the folder definitions and show that for this special case of the problem there exists a polynomial-time algorithm that delivers an optimal solution. Thus, we have identified an interesting special case of OR view graphs and presented an optimal algorithm for the view-selection problem considered in [Gup97] under the optimization criteria addressed in this paper.

Generalization of the above problem from one-dimensional selection queries to d-dimensional query and view hyper-rectangles doesn't have any better than the $O(\log n)$ approximation algorithm. The d-dimension selection queries case can be shown to be NP-complete through a reduction from 3-SAT. In fact, the problem is a more general version of the classical age-old problem of covering points using rectangles in a 2-D plane [FPT81], for which nothing better than an $O(\log n)$ approximation algorithm is known.

4.3 Queries as Selections + Unions over Views

In this section, we look at some special cases of the general folder-selection problem, while allowing both selection and union operators for computation of a folder from other materialized folders. The use of both operators introduces hyperedges in the query-view graph. As mentioned before, the general folder-selection problem involving hyperedges is intractable, hence we take advantage of the restricted model of the folder definitions in designing approximation algorithms.

The folder-selection problem with union and selection operators is a special case of the view-selection problem in AND-OR view graphs considered in [Gup97], with different optimization criteria and constraints. Gupta [Gup97] fails to give any approximation algorithms for the general view-selection problem in AND-OR graphs. We take advantage of the special nature of our problem, and present polynomial-time approximation algorithms.

4.3.1 One-dimensional Selection/Union Queries

Consider an ordered domain D, and let folders be ranges over D. In other words, views and queries can be represented as intervals over D and a query interval can be answered using views $v_1, \ldots, v_l$ if the union of the view intervals covers the query interval completely. There is a query-cost associated with each such pair, which is the cost incurred in computing q from $v_1, v_2, \ldots, v_l$.

Problem (One-dimensional Selection/Union Queries) *Given a set of interval views V and interval queries Q in an ordered domain D, select a minimum weighted set of interval views M such that each query $q \in Q$ has a set of views $v_1, v_2, \ldots, v_l \in M$ that completely cover q and the query-cost associated is less than T_q.*

Consider the following cost model. In addition to a weight associated with each view, let there also be a cost $C(v)$ associated with each view. And the cost of computing a query q using a set of views $\{v_1, v_2, \ldots, v_l\}$ that cover the query q is defined as $\sum_{i=1}^{l} C(v_i)$, i.e., the sum of the costs of the views used. The above cost model is general enough for all practical purposes.

The problem of one-dimensional selection/union queries with the above cost model can be shown to be NP-complete through a reduction from the NP-complete Partition [GJ79] problem as follows. Consider an instance $A = \{a_1, a_2, \ldots, a_n\}$ of Partition. The instance A is in Partition if and only if there exists an index set $I \subset \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} a_i = (\sum_{i=1}^{n} a_i)/2$. Given an instance A of the Partition problem, consider the corresponding instance $G(A)$ of the one-dimensional selection/union queries as shown in Figure 1. The set Q consists of only one query q with the query threshold $T_q = (\sum_{i=1}^{n} a_i)/2$, and the set V consists of $2n$ views v_{ji} for each $1 \leq i \leq n$ and $j = 1, 2$.

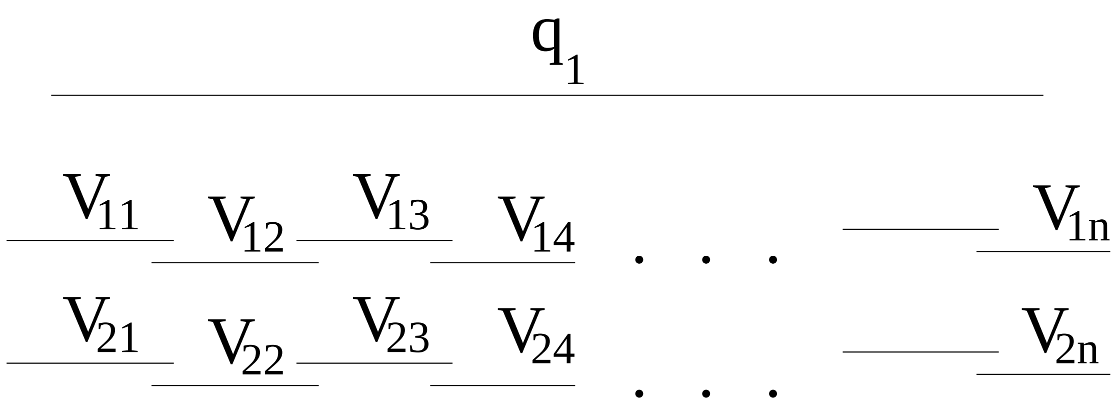

Figure 1: The instance $G(A)$ of one-dimensional selection/union queries

The costs and weights associated with the views are as follows: $C(v_{1i}) = a_i$ and $W(v_{1i}) = 0$ for all $1 \leq i \leq n$. Also, $W(v_{2i}) = a_i$ and $C(v_{2i}) = 0$ for all $1 \leq i \leq n$. It is not difficult to see there is a set of views of weight less than $(\sum_{i=1}^{n} a_i)/2$ that covers the query q within query-threshold T_q if and only if $A \in$ `Partition`.

If we restrict our attention to the *index cost model* where the cost incurred in computing a query covered by l views is l units (a special case of the general cost model above, where each $C(v) = 1$), we show that there exists an $\mathrm{O}(m^{k-1}n^2)$ dynamic programming solution, where m is the maximum overlap between the given queries and k is the maximum individual query-cost threshold. The index cost model, where the cost of answering a query q using l views is l units, is based on the following very reasonable implementation. If all the materialized views are indexed along the dimension D, then the cost incurred in computing the query is proportional to l, the total number of index look-ups. Note that the query-costs associated with the edges may not be the actual query costs but could be the normalized query-cost "overheads". We now show the $O(n^2m^{k-1})$ solution to the problem.

Dynamic Algorithm Solution Let $e_1, e_2, \ldots, e_n$ be in increasing order the right end-points of the view intervals. We maintain a set of $\binom{m}{k-1}$ solutions for each interval $[1, e_i]$. Thus, we maintain a table of $O(nm^{k-1})$ solutions. Computation of each solution from previous solutions would take $O(n)$ time yielding the desired time complexity.

Consider the point e_1 and let $q_1, q_2, \ldots, q_m$ be the queries intersecting it. We represent a solution at $[1, e_i]$ by $S^i_{i_1,i_2,\ldots,i_{k-1}}$ where $1 \leq i_1 \leq i_2 \ldots i_{k-2} \leq i_{k-1} \leq m$. Note that there would be at most $\binom{m}{k-1}$ such solutions at $[1, e_i]$. We defined q_{ji} to be a new query interval such that $q_{ji} \subseteq q_j$ and the right endpoint of q_{ji} is e_i. See Figure 2. A solution $S^i_{i_1,i_2,\ldots,i_{k-1}}$ represents the minimum weighted set of views that answer all the queries that have their rightmost points less than e_i within their respective thresholds and queries $q_{1i}, q_{2i}, \ldots, q_{mi}$ with their query thresholds defined as follows. The subscript $i_1, i_2, \ldots, i_{k-1}$ of the solution signifies that the solution corresponds to the case when the query thresholds of $q_1, q_2, \ldots, q_{i_1}$ is 1, of $q_{i_1+1}, \ldots, q_{i_2}$ is 2, $\ldots$, and of $q_{i_{k-1}}, \ldots, q_m$ is k. That is, query threshold of q_{ji} is l iff $i_{l-1} < j \leq i_l$, where i_0 is considered 0 and i_m is k. The final solution

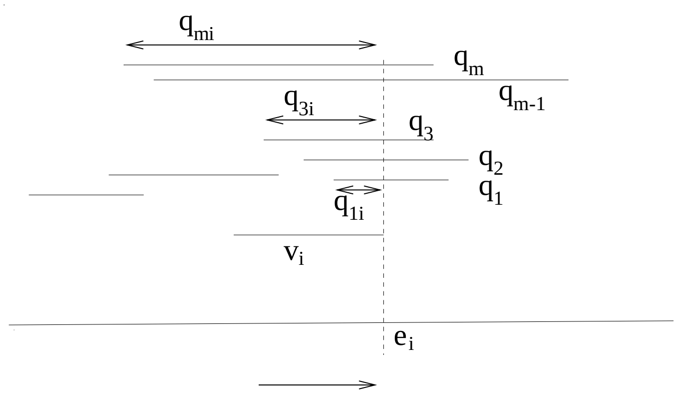

Figure 2: Setting up the set of solutions for the dynamic programming approach

is given by S_0^n, assuming that no query interval intersects v_n, else, there would be no way to cover that query.

It is not difficult to see that $S^i_{i_1,i_2,\ldots,i_{k-1}}$ can be computed in $O(n)$ time by considering each view in V that intersects e_i. We omit the details here.

Average Query Cost Constraint A relatively easier problem in the context of the above cost model is when the constraint is on the *total* (or, *average*) query cost instead of having a threshold on each individual query. For such a case, there exists an $O(kn^3)$ time dynamic programming algorithm that delivers a minimum-weighted solution, where $k(\leq n)$ is the average query-cost constraint. The dynamic approach here works by maintaining for each interval $[1, i]$ a list of k solutions, where the j^{th} solution corresponds to the minimum-weighted set of views that covers the queries in $[1, i]$ under the constraint that the total query cost incurred is less than j.

4.3.2 Multi-dimensional Selection/Union Queries

Consider the problem where queries and views are d-dimensional ranges. In other words, views and queries can be represented as hyper-rectangles in a d-dimensional space and a query interval can be answered using views $v_1, \ldots, v_k$ if the union of the view hyper-rectangles covers the query hyper-rectangle completely. We wish to select a minimum-weighted set of views such that all queries are covered. This simple version has no threshold constraints.

The above problem is NP-complete even for the case of two dimensions. We present here a polynomial-time (in n) $O(d \log n)$ approximation algorithm. The space of hyper-rectangular queries can be broken down into $O((2n)^d)$ elementary hyper-rectangles as follows. Projection of the n hyper-rectangles onto each of the d dimensions results in $2n$ points in each of the d dimensions. It is easy to see that there are $(2n)^d$ basic hyper-rectangles and any hyper-rectangle formed on these $2n$ co-ordinates in each dimension can be represented as union of these $(2n)^d$ basic hyper-rectangles. Thus, the problem of covering the query hyper-rectangles can be reduced to covering the elementary hyper-rectangles with minimum-weighted set of views, which is equivalent to a set cover instance having $O((2n)^d)$ elements; this has an $O(d \log n)$ approximation algorithm.

	Cost Constraint	1-dimension	d-dimension	General Case
Selection	—	$O(n^2)$ exact	$O(\log n)$ approx.	$O(\log n)$ approx.
Selection/Union	No Thresholds	$O(n^2)$ exact	$O(d \log n)$ approx.	?
	Individual Thresholds	$O(m^{k-1}n^2)$ exact[7]	?	?
	Total Cost Threshold	$O(kn^3)$ exact	?	?

Table 1: Summary of the Algorithmic Results on the Folder-Selection Problem

4.4 Related Work

The folder-selection problem is similar to the view-selection problem defined in [Gup97]. The view-selection problem considered there was to select a set of views for materialization to minimize the query response time under the disk-space constraint. The key differences between the two problems are the different constraint and the minimization goal used.

Previous work on the view selection problem is as follows. Harinarayan et al. [HRU96] provide algorithms to select views to materialize for the case of data cubes, which is a special case of OR-graphs, where a query uses exactly one view to compute itself. The authors in [HRU96] show that the proposed polynomial-time greedy algorithm delivers a solution that is within a constant factor of the optimal solution. Gupta et al. [GHRU97] extend their results to selection of views *and* indexes in data cubes. Gupta [Gup97] presents a theoretical formulation of the general view-selection problem in a data warehouse and generalizes the previous results to general OR view graphs, AND view graphs, OR view graphs with indexes, and AND view graphs with indexes.

The case of one-dimensional selection queries considered here is a special case of the view-selection problem in OR view graphs for which we provided a polynomial time algorithm that delivers an optimal solution. Similarly, the case of one-dimensional selection/union queries is a special case of the view-selection problem in AND-OR view graphs ([Gup97]), which we observe can be solved optimally in polynomial-time for a reasonable cost model.

In the computational geometry research community, to the best of our knowledge, the specific problems mentioned here haven't been addressed except for the preliminary work done on rectangular covers [FPT81].

5 Conclusions

In this paper, we have identified and addressed two complementary problems that arise in a messaging system, which is a special kind of a data warehouse, where folders are defined as selection views over the message store.

One of the most important tasks of a messaging system concerns the efficient incremental maintenance of eagerly maintained folders. The incremental maintenance problem for our model

[7] Here, m is the maximum number of queries that overlap and k is the maximum individual (or average) query-cost constraint.

of folder definitions presented here is a more general version of the point-location problem. The problem in its full generality has not been discussed before. We designed an I/O and CPU efficient algorithm for this problem, based on external segment trees and tries.

An important design decision that a messaging system needs to make is the choice of eagerly maintained (materialized) folders. We formulated the folder-selection problem as one of selecting a minimum-weighted set of folders such that every other folder can be computed from the selected set within some threshold cost. We discussed various special cases of the general folder-selection problem in the context of a messaging system with folders defined as selection views. In this context, we presented various exact/approximation algorithms, and complexity hardness results.

We have thus identified several interesting problems in the area of data warehousing, where techniques from computational geometry and complexity theory are helpful, which should be of interest to the database theory community. There are still quite a few open theoretical questions. Noteworthy among them are:

- Is there a polynomial-time approximation algorithm for the general folder-selection problem? Can we say anything about the non-approximability of the problem?
- Is there a polynomial-time (in n and d) approximation algorithm for the problem of covering rectangles using rectangles, even for the simple case when there are no constraints?

References

[ABSS93] S. Arora, L. Babai, J. Stern, and Z. Sweedyk. The hardness of approximate optima in lattices, codes, and systems of linear equations. In IEEE, editor, *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 724–733, Palo Alto, CA, November 1993. IEEE Computer Society Press.

[AL95] Sanjeev Arora and Carsten Lund. Hardness of approximations. Technical Report TR-504-95, Princeton University, Computer Science Department, December 1995.

[Cha83] B. Chazelle. Filtering search: A new approach to query-answering. In *Proceeding of the IEEE Conference on Foundations of Computer Science*, pages 122–132, 1983.

[Chv79] V. Chvatal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[Cri94] M. Crispin. Internet message access protocol: Version 4. Technical report, December 1994. Request for Comments 1730. Available from ftp://ds.internic.net/rfc/rfc1730.txt.

[Cro82] D. H. Crocker. Standard for the format of ARPA Internet text messages. Technical report, August 1982. Request for Comments 822. Available from ftp://ds.internic.net/rfc/rfc822.txt.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[Dom] *Lotus Domino*. http://www.lotus.com/domino/.

[Ede83a] H. Edelsbrunner. A new approach to rectangle intersections, Part I. *International Journal of Computer Mathematics*, 13:209–219, 1983.

[Ede83b] H. Edelsbrunner. A new approach to rectangle intersections, Part II. *International Journal of Computer Mathematics*, 13:221–229, 1983.

[EM81] H. Edelsbrunner and H.A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13(4):177–180, 1981.

[FPT81] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12(3):133–137, June 1981.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–500, 1960.

[GJ79] M. R. Garey, D. J. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection in OLAP. In *Proceedings of the International Conference on Data Engineering*, Burmingham, U.K., April 1997.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, June 1995.

[GM97] M. Goldwasser and R. Motwani. Intractability of assembly sequencing: Unit disks in the plane. In *Proceeding of the Workshop on Algorithms and Data Structures*, August 1997.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece., January 1997.

[HCKW90] E. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 271–280, 1990.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

[HRU96] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD International Conference of Management of Data*, Montreal, Canada, June 1996.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse*. QED Publishing Group, Boston, Massachusetts, 1993.

[KRVV93] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, and J.S. Vitter. Indexing for data models with constraints and classes. In *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems*, pages 233–243, Washington, D.C., May 1993.

[OIO] *Oracle Inter Office*. http://www.interoffice.net/.

[Ros91] M. T. Rose. Post office protocol: Version 3. Technical report, May 1991. Request for Comments 1225. Available from ftp://ds.internic.net/rfc/rfc1225.txt.

[RS94] S. Ramaswamy and S. Subramanian. Path caching: A technique in optimal external searching. In *Proceedings of the Thirteenth ACM Symposium on Principles of Database Systems*, pages 25–35, Minneapolis, Minnesota, May 1994.

[RS95] S. Ramaswamy and S. Subramanian. The p-range tree: A new data structure for range searching in secondary memory. In *Proceeding of the ACM Symposium on Discrete Algorithms*, pages 378–387, 1995.

[Sam89a] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1989.

[Sam89b] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[SIMS] *Sun Internet Mail Server*. http://www.sun.com/sims/.

[VV96] D.E. Vengroff and J.S. Vitter. Efficient 3-D searching in external memory. In *Proceeding of the ACM Symposium on Theory of Computing*, pages 192–201, Philadelphia, PA, May 1996.

[WGL+96] J. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 1996.

[Wid95] J. Widom. Research problems in data warehousing. In *Proceedings of the Fourth International Conference on Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, November 1995.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, May 1995.