Merging Multiple Data Streams on Common Keys over High Performance Networks

Marco Mazzucco, Asvin Ananthanarayan,
Robert L. Grossman*, Jorge Levera, and Gokulnath Bhagavantha Rao
Laboratory for Advanced Computing (M/C 249)
University of Illinois at Chicago

April, 2002; Revised June, 2002

This is a draft of a paper from the Proceedings of the IEEE/ACM SC2002 Conference, 2002, IEEE Computer Society, page 67.

Abstract

The model for data mining on streaming data assumes that there is a buffer of fixed length and a data stream of infinite length and the challenge is to extract patterns, changes, anomalies, and statistically significant structures by examining the data one time and storing records and derived attributes of length less than N. As data grids, data webs, and semantic webs become more common, mining distributed streaming data will become more and more important. The first step when presented with two or more distributed streams is to merge them using a common key. In this paper, we present two algorithms for merging streaming data using a common key. We also present experimental studies showing these algorithms scale in practice to OC-12 networks.

*Point of Contact: grossman@uic.edu, http://www.ncdm.uic.edu. R. Grossman is also with Two Cultures Group.

1 Introduction

Data mining is the semi-automatic extraction of patterns, changes, anomalies and statistically significant structures from large data sets. In general an unspoken assumption has been that the data has been *at rest* and algorithms can access the data as often as required. Of course, the goal is to devise data mining algorithms with the lowest complexity.

Recently, data mining has begun to be applied to data *in motion*. To describe this more precisely, fix a record structure x containing attributes of interest and a window (buffer) of size N records. The data consists of a stream x_1, x_2, x_3, ... of records, where N is much less than the length of the stream. The goal is still to extract patterns, changes, and anomalies from the data, but the assumption is that the data is seen only once and one is only able to retain records and derived attributes which fit into the window.

This point of view is motivated by a number of applications, including:

- A large network can contain dozens to hundreds of routers and switches, each of which produce large data streams. The analysis of the network depends upon analyzing these data streams.

- Wireless sensors are becoming commoditized and applications are being planned in which thousands of sensors will produce streaming data, only a small fraction of which will be stored.

- Routing, filtering and selection systems for data flowing from satellites are faced with large volume data streams which are sent to other systems for processing and analysis.

- Web based infrastructures for data are emerging which support remote data analysis and distributed data mining. These include data grids, semantic webs, and data webs. Interactive exploration of remote data and distributed data mining using multiple data sources are both naturally modeled by streaming data.

Our concern is with the problem of merging two data streams using a common key. This is one of the fundamental operations required for remote data analysis and distributed data mining. Merges are one of the fundamental operations underlying the management of local relational data. As data grids, data webs, semantic webs and related infrastructures mature, merging remote and distributed streaming data will become just as fundamental.

In this paper, we introduce this problem, introduce two algorithms for attacking it, and describe some experimental studies showing that our algorithms and implementations scale to high performance OC-12 networks. The experimental studies use a data web infrastructure we have developed called DataSpace.

Our approach incorporates the following four significant innovations:

1. Most prior work processing streaming data has focused either on developing versions of data mining algorithms, such as clustering or trees [18, 4] for streaming data, or on continuous query systems, in which the goal is to adapt query optimization techniques, such as query plans, to queries on streaming data [21, 3]. Our work is concerned with merging *high volume* data streams for real time, interactive data analysis of remote and distributed data sets. To achieve this scalability we layer our windowed merges over specialized libraries for high performance data transport [7, 8]. We present the first experimental evidence we are aware of for the scalability of windowed operations on OC-3 and OC-12 data streams. We believe that integrating specialized operations on streaming data with specialized data transport protocols will enable the remote analysis and distributed mining of even very large remote data sets.

2. We introduce best effort δ-merges which provide higher performance for merging high volume data streams at the expense of not matching all the records. With this approach, we can drop δ percent of the records, which is acceptable for many applications, such as the initial interactive exploratory data analysis and data mining of remote and distributed data.

3. We also introduce best effort ϵ, δ-merge which, in addition to being a δ-merge, assumes that a record may be merged not only with an exact match but with any record whose key is within an ϵ distance. This is acceptable for certain applications, such as when the keys represent latitude, longitude, and time, or similar real valued dimensions. The two algorithms presented in this paper are examples of ϵ, δ-merges.

4. Prior work has assumed that streaming data is produced by sensors and other data feeds which produce streaming data but otherwise have limited functionality [21, 20, 3]. Our interest is in data webs, data grids and semantic webs in which remote data servers provide streaming data. We assume that the data sources are data servers which can

support R-trees and other index structures which can be queried by the streaming data client. In other words, we introduce the idea of merging high volume data *indexed* data streams. We are particularly interested in multi-dimensional indexes which arise when working with gridded data [9].

The paper is organized as follows. In Section 2 we describe background and related work. In Section 3 we describe the set up and the main ideas. Section 4 introduces the two algorithms and Section 5 gives some experimental results. Section 6 is the summary and conclusion.

2 Background and Related Work

Versions of the basic algorithms for data mining, including algorithms for clustering and trees, have recently been proposed for streaming data. The papers by O'Callagan et. al. [18] and Guha et. al. [12] describe algorithms for clustering streaming data. The paper by Domingos and Hulten [4] describes an algorithm for building trees on streaming data.

During the past several years, the optimization of SQL-type queries for streaming records has been studied. These are sometimes called continuous query systems. Continuous query systems were proposed by Terry et. al. [21]. The OpenCQ system supports actions consisting of four tuples containing a SQL-like query, a database trigger condition, a start condition, and an end condition [15]). NiagaraCQ is another continuous query system which is targeted at efficiently evaluating collections of complex SQL queries over streaming data, such as stock market feeds or news feeds [3]. These systems are based upon the idea of building query plans by extracting common subexpressions, rearranging selections and projections, and related optimizations given a collection of SQL queries. Our focus is the efficient merging of a small number of very high volume data streams for remote interactive data analysis and distributed data mining.

Important differences between querying persistent and streaming data were discussed in [20], including using windows for querying streaming data. The idea of improving the efficiency of merges over very large databases using windows was studied by DeWitt et. al. [2].

A key component in merging distributed data streams is sorting the two streams by their common keys. Previous work on sorting data streams was conducted by Juggle [19]. In this paper they outline a method by which data can be retrieved from a distributed data repository and served in a partially ordered fashion. Assumptions of the data being indexed are made.

Their focus is the interactive control of queries to streaming data. Our focus is on non-interactive high performance merging of distributed streams.

3 Streaming Merges

In this section, we describe some of the basic ideas.

Fix an ordered index $i \in \mathcal{I} = \{1, 2, 3, \ldots\}$. Given a stream (k_i, x_i) and another stream (k_j, y_j) of records sharing a common key k_i, the *merge* is defined to be the stream (k_i, x_i, y_i). Here k_i are indices called keys and x and y are records which in general do not share any attributes, although they may.

Fix an integer N, which we will use to specify the window size. Given two windows of size N extracted from each of the streams, the number of keys k_i which match can vary from 0 to N. Given a sequence of windows w_1, w_2, ..., let κ_1, κ_2, ..., denote the percentage of matches. To measure the completeness of the merge, we take the moving average on the sequence κ_i. Let m be an interger such that $m > 0$, where m represents the number of windows we compute our average over. Let n be any number, such that $n \geq m$, where n represents the last window to be included in the average. Then, the moving avergage $\Delta_{m,n}$ is given by:

$$\Delta_{m,n} = \frac{\sum_{i \in [(n-m+1),n]} \kappa_i.}{m}.$$

In general, merging all the records in two streams will not be possible. Given two streams, drop one of the records if the corresponding record is not present. A *best effort delta-merge* is a merge of the two streams in which, for some $m > 0$, for all $n \geq m$, $\Delta_{m,n} < \delta$.

Rather than drop a record from a stream if the corresponding record is not present, we can approximate it with a nearby record. Define an approximate ϵ-merge to be a merge in which a record (k_i, x_i) can be merged with any record (k_j, y_j) where $|k_i - k_j| \leq \epsilon$.

Best Effort ϵ, δ-merge. Fix a window of size N records. For a fixed δ and ϵ we define a best effort ϵ, δ-merge to be a δ-merge in which matches are ϵ-approximate.

A sequence of sliding windows with index i and a best effort matching gives a sequence of matching rates κ_i. To improve κ_i, we can lower the transmission rate of the windows, increase the window size, or change the algorithm which merges the streams. Let $\kappa_{\max}$ denote the matching rate when the window size is sufficiently large to store all of the input stream

(which of course violates the basic assumption which distinguishes mining streaming data from mining data at rest).

Our interest is in algorithms in which N is much less than the length of the data stream but in which for all m and n, $\Delta_{m,n}$ is close to $\kappa_{\max}$.

4 Best Effort ϵ, δ-Merges

In this section, we describe two algorithms for best effort ϵ, δ-merges.

The first algorithm assumes that the data streams are partially sorted. Since this is often the case when the data is being continuously generated, we refer to this algorithm as the continuously generated data merge, or CGM algorithm for short. Examples of possible applications of CGM merges would be satellite images of different modalities (merged by latitude/longitude) or route dumps (merged on source addresses).

The second algorithm makes no assumption on the order of the data, but assumes that the data is indexed by a data server at the *data source* by an R-tree. For some applications involving continuously generated data, it may not be possible to support R-trees, but for other applications such as data webs and data grids, as mentioned above, this is a natural assumption. We will refer to this algorithm as the R-tree merge algorithm or RTM for short.

Before describing the first algorithm, we would like to make more precise what we mean by sorted. We borrow the term Universal Correlation Key or UCK from [9] to refer to a key or keys attached to distributed columns of data. Just as URLs are locators attached to distributed documents, UCKs are locators attached to distributed columns of data — essentially global keys which support merges so that one column of data can be correlated with another column of data. Our goal is to merge one or more distributed columns of data by their UCKs. Remote data sets may contain more than one UCK. When there are two data sets that can be merged on multiple UCKs, in order to merge the data, the data must be sorted lexigraphically on the UCKs. Thus, when we say the data is sorted in this context, we mean that the data is sorted lexigraphically on multiple UCKs. Furthermore, for a merge to take place, both data sets must have similar lexigraphical ordering. Our experiments use grid data from NCAR [17] with multiple UCKs representing latitude, longitude and time. Notice that for grid data, ϵ-merges can provide meaningful approximations for certain applications.

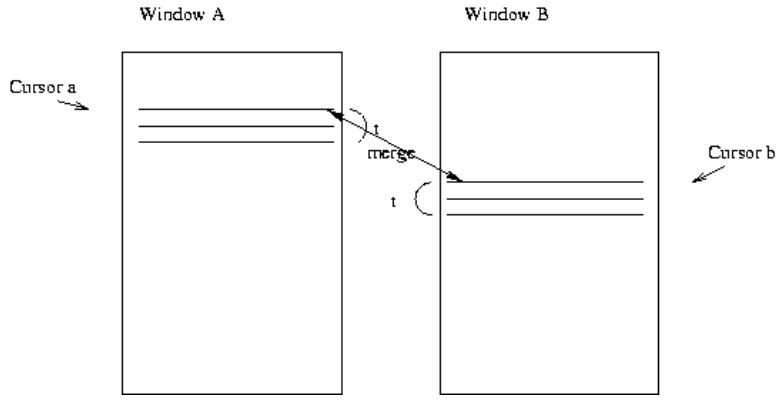

Figure 1: CGM merge with tolerance ϵ.

4.1 CGM Algorithm

In the CGM algorithm we assume the data is partially presorted. Without loss of generality, assume there are two data streams, A and B, being drawn into a client in approximately ascending order and we are trying to merge on one UCK. The CGM algorithm depends upon three parameters: a parameter N determining the number of records in a window, that is used to buffer the streaming data, and two parameters N_a and N_b which are used to determine the minimum amount of new data records which are read into the window after each processing step. Fix integers N_a, N_b and N such that $N_a, N_b < N$ and a tolerance ϵ. The steps of the algorithm are as follows:

1. The client grabs some fixed number of records N, from both stream A and stream B and places them in window A and window B respectively (each has room for exactly N records).

2. A sort is done on both windows (since the data is only partially sorted).

3. A merge is then attempted. This process takes at most $2N$ steps. In the first step, cursor a is on the first record of the window A and cursor b is on the first record of the window B. In step n, assume that cursor a is on record $m \leq n$.

   (a) If cursor b points to a UCK whose value is greater than the UCK value pointed to by cursor a plus ϵ than we move cursor a down one record.

(b) If cursor b points to a UCK value whose value is within ϵ of the UCK value pointed to by a, then merge the two records and move a down so that it points to the least UCK value which is greater than ϵ plus the value which was just merged. Similarly we move cursor b down.

(c) Finally, if a points to a UCK value which is greater than the UCK value pointed to by b plus ϵ, we move cursor b down one record.

4. An insertion of new data into the window takes place if the cursor is on the N'th record. A general illustration of the process is shown in Figure 2. Without loss of generality, suppose cursor a reaches the N'th record first and the data is being received in approximately ascending order. (Note, a similiar process occurs if cursor b reaches the N'th record first.)

   (a) Let M be the number of successful merges in the previous iteration. Define l_a (respectively l_b) to be the greatest of M and N_a (respectively N_b). We refer to N_a and N_b as the *least window increment value*. It is the least amount of records that the window moves forward in one step. Without N_a and N_b, the algorithm could deadlock if sufficiently few matches occur when processing the window.

   (b) We place l_a new records into the window A. The size of the window remains unchanged.

   (c) The new records are first placed in the memory locations of the successfully merged data records of window A and then, if $N_a - M > 0$, in the memory locations of the first $N_a - M$ records in window A. Thus, by replacing the first $N_a - M$ records, we use the fact that the window is sorted to replace only the records which have been in window A for the longest period of time.

We continue this process until one or both data streams are exhausted. Note the purpose of N_a and N_b is to advance the window by a least set number of records, independent of the number of records matched. Clearly the complexity of this algorithm is $n\log(n) + n$. In practice, for partially sorted data, the cost can be much less.

As an example, let $N = 8$ and $\epsilon = 2$ and suppose in window A we have $\{6, 7, 8, 9, 10, 11, 20, 21\}$ and in window B, $\{5, 13, 14, 15, 16, 17, 18, 21\}$. Then the resulting merge would be
$\{(6, 5), (11, 13), (20, 18)\}$.

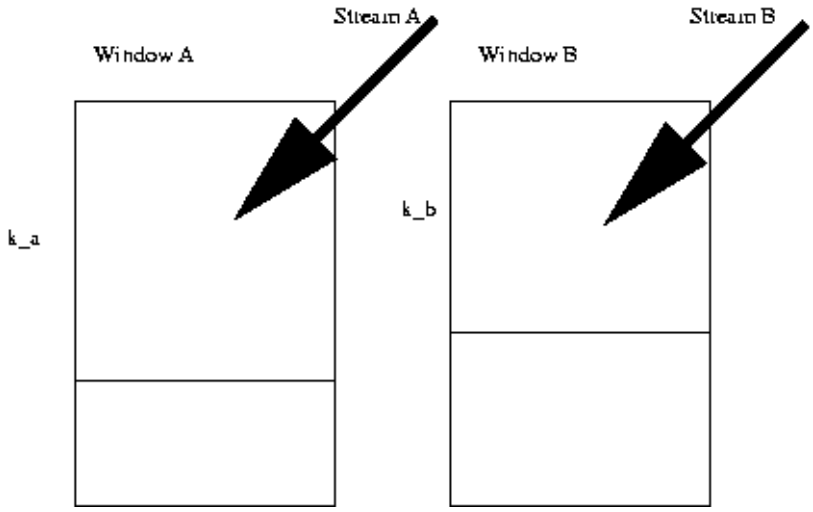

Figure 2: CGM insert operation during low match rates.

4.2 RTM Algorithm

In the second algorithm we utilize R-trees. We assume that the remote data server supports R-trees. R-trees allow one to query data by range: one can specify n columns and use the R-tree to retrieve efficiently the corresponding n-cube of records.

As an example, for grid data using latitude, longitude, and time as UCKs, a range query might return all precipitation data between the equator and 23 degree latitude, 35 to 45 degree longitude, for the past six months from one data server and overlay this with data from another data server containing records for floods during the same period and for the same area.

We assume that the data client is drawing data from two data streams, stream A and stream B. We assume that stream B can access data via range queries. The steps of this algorithm are as follows:

1. A block of data (the first N records) is grabbed from stream A.
2. Determine the max and min of each UCK on which the merge will take place and do a query on stream B, requesting for the n-cube of data corresponding to the boundary region of the data in window A. Retrieve the first N records, and place them into window B.
3. The sort and merge method of the CGM algorithm (4.1 steps 2 and

3) on windows is employed.

4. In the insertion step:

   (a) If cursor a reaches the N'th record first, we clear window A and go to step one.

   (b) If cursor b reaches the N'th record first, we clear window B. Then we query the R-tree from stream B for the symmetric difference of the n-cube described by the previous window B and the n-cube described by existing window A (window A remains unchanged). If the query is nonempty, we place exactly the number of records which fit in to window B from stream B. If the query is empty, we clear window A and go to step 1.

The process continues until stream A is exhausted. As an example, let $N = 5$ and $\epsilon = 0$ and suppose window A contains $\{2, 4, 6, 8, 10\}$. After a query on stream B, window B contains $\{2, 3, 4, 5, 6\}$. A merge is than conducted, and a query of the symetric difference of window A and window B is done on stream B, which results of the insertion of $\{7, 8, 9, 10\}$ into window B. The result of the merge is $\{(2, 2), (4, 4), (6, 6), (8, 8), (10, 10)\}$.

5 Design and Implementation

We implemented these algorithms using a data web we have developed called DataSpace [9]. DataSpace uses a protocol called the DataSpace Transfer Protocol or DSTP. We have developed open source DSTP clients and servers. For these experiments we modified the DSTP server to support the CGM and RTM algorithms. We also developed a specialized client to merge two streams and measure several parameters described below.

The implementation uses two buffers, as shown in Figure 3. The incoming data is appended into one memory location, while a separate thread of the software copies blocks of data (windows) from this memory location into another memory location where the above algorithms take place, thus making the merge software nonblocking provided that the processing time for the merge is sufficiently fast. A query for the next data block can be issued while a merge is being conducted on the current data block. In the case of RTM, by the design of DSTP, a query can be issued to a DSTP Server, while data is being streamed in from a previous query. In the case of CGM, data is requested in blocks from a DSTP Server or simply streamed in as the

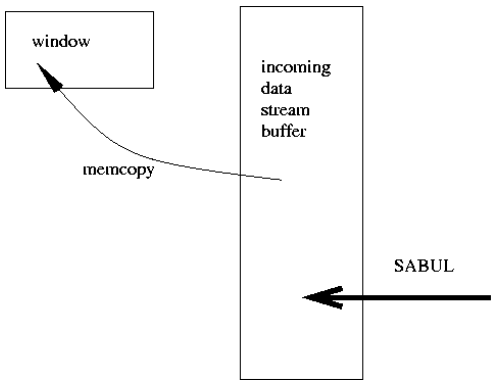

Figure 3: A diagram illustrating dual buffering.

data is being generated. In all cases, the data stream can be nonblocking by the parallel design of both the DSTP Server and DSTP protocol.

We integrated R-Trees from GIST [13] into our DSTP Server for these experiments. This is a light weight library which is highly customizable.

6 Experimental Results

We used an OC12 (622Mb/s) WAN to test these algorithms. We present results for both RTM and CGM. We also took separate measurements for network transfer and merges, thus giving us a better feeling of how much the network is a bottleneck for both algorithms.

For the CGM algorithm, we varied several variables: the window size, the least window increment value and the percentage randomness of the data stream. We only considered the case when $N_a = N_b$. To avoid confusion, we refer to this value as K. The percentage randomness of the data stream was measured by taking an ordered data file, and reordering a fixed percent of the file. For the RTM algorithm, we varied the same variables except for window increment value, which plays no part in this algorithm. The data file used in all of these experiments was gridded atmospheric data obtained from NCAR [17]. The tolerance for all these experiments was fixed.

The experimental setup was as follows. Data servers were located in Amsterdam and Chicago and connected via an OC-12 network. The merge was done on a separate machine in Chicago. The machine in Amsterdam was a dual P4, 1700 Mhz, with 512M ram. The machines in Chicago were dual PIIIs, 1000Mhz, with 512M ram. The machines were running Linux,

with the 2.4.x kernels. The network traffic was over SurfNet.

For the following two tables, two data files containing 750,000 records were used. The data files were 21MB each, so that the amount of data streamed to the machine where the merge algorithm was run was 42MB. There are two things that should be observed. Currently the merge client is not threaded, so we expect the Total Time to be near the maximum of the Merge Time and Fetch Time once the client is threaded. Also, the data stream used a high performance data transport library called SABUL [8]. Previously measured performance with SABUL has been better. We suspect that there are issues with how the merge application is using the SABUL library.

In Table 1 we show the values for a window size of 5000 records, while in Table 2 we show the values for 10000 records.

K	Rand %	Match %	Fetch Time *sec*	Merge Time *sec*	Total Time *sec*
1000	2	100	0.639	0.587	1.228
2000	2	100	0.639	0.593	1.233
4000	2	100	0.639	0.596	1.236
1000	10	99	0.638	1.068	1.71
2000	10	99	0.637	1.067	1.699
4000	10	87	0.638	0.576	1.216
1000	20	99	0.637	1.1034	1.741
2000	20	99	0.637	1.086	1.723
4000	20	72	0.638	0.652	1.291
1000	33	99	0.64	1.445	2.085
2000	33	89	0.64	1.288	1.928
4000	33	48	0.639	0.669	1.309

Table 1. CGM with 750,000 records and window size of 5,000 records.

K	Rand %	Match %	Fetch Time *sec*	Merge Time *sec*	Total Time *sec*
2000	2	100	0.638	0.647	1.285
4000	2	100	0.639	0.646	1.285
8000	2	100	0.638	0.64	1.279
2000	10	99	0.637	0.767	1.404
4000	10	99	0.638	0.769	1.407
8000	10	87	0.638	0.643	1.281
2000	20	97	0.637	1.268	1.905
4000	20	97	0.637	1.261	1.899
8000	20	72	0.638	0.691	1.329
2000	33	99	0.636	1.477	2.114
4000	33	92	0.64	1.292	1.932
8000	33	49	0.638	0.727	1.365

Table 2. CGM with 750,000 records and window size of 10,000 records.

Finally we present our results for the RTM algorithm. This was done on a file containing 100,000 records. Recall that K has no meaning in RTM. The window sizes of both 10,000 and 5,000 were used.

Win Size *records*	Rand %	Match %	Total Time *sec*
5000	2	86	.475
5000	10	74	.472
5000	20	70	.478
5000	33	69	.471

Table 3. RTM with 100,000 records and window size of 5,000.

Win Size *records*	Rand %	Match %	Total Time *sec*
10000	2	92	.485
10000	10	82	.487
10000	20	79	.484
10000	33	71	.482

Table 4. RTM with 100,000 records and window size of 10,000.

7 Discussion

As the code is implemented now, our CGM algorithm can merge nearly ordered data (data which has less than or equal to two percent out of order

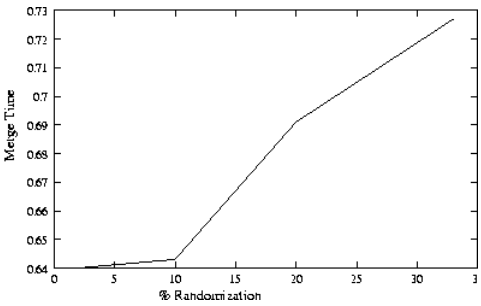

Figure 4: CGM for K=8000

records) at a rate of 34MB/s. As mentioned, the merge client software as of yet is not threaded, so we hope to obtain times of around 60MB/s once threading has been implemented. The Fetch Times indicate a peak aggregate (two streams) bandwidth of around 520Mb/s. We still have to determine if the bottleneck is a buffering problem between SABUL and our merge client or DSTP Server performance.

In general the results for the CGM algorithm were as expected. With increased disorder in the data, merge times increased. A graph of this phenomena for a fixed value of K can be seen in Figure 4. Figure 5 shows percentage of successful merges against time for merge. The data stream for this graph had 33% of the records out of order. The varying degrees of successful merges were obtained by varying K. As expected, if K is greater than the expected number of successful merges per window, the total number of successful merges decreases and the merge time decreases. If K was less than the expected number of merges per window, the merge time increased and the number of total successful merges increased. This follows from the following two facts: For smaller values of K, records which have not been merged tend to stay in the window until they are merged, while for larger values of K they are discarded. Records that have not been merged stay in the window for a period determined by K and the percentage matched in the CGM algorithm.

As expected, since R-trees do not retrieve the data in sorted order, the degree of randomness in the data stream does not effect the time required for the streaming merge. The times for this algorithm are slightly higher than that of the CGM algorithm due to the increased time required to retrieve data using an R-tree and the fact that the sort takes longer than for the partially sorted data occurring with the CGM.

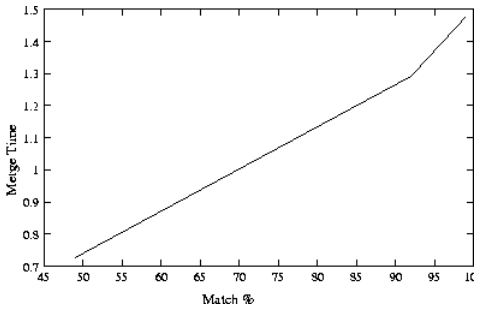

Figure 5: CGM for 33 % Randomization.

8 Summary

In this paper, we introduced algorithms for merging two or more data streams by a common key and showed experimentally that our algorithm scales to wide area OC-12 networks.

As data webs, data grids, and semantic webs grow more common, merging distributed columns of remote and distributed data will be a core component of many algorithms for remote data analysis and distributed data mining. As high bandwidth networks grow more common, scalable algorithms for merging remote and distributed data will enable the analysis and mining of remote data as if it were local.

With the best effort ϵ, δ-merge algorithm introduced in this paper, two or more streams are merged by their common key with records being dropped if they do not arrive in a synchronous fashion within a window of fixed size. The goal is to keep the percentage of dropped records less than δ. For many applications, best effort ϵ, δ-merges are good enough. Moreover, best effort merges of this type are consistent with a computational model in which the streams are much longer than a window of N records which must hold all retained raw and derived data. We call this algorithm the Continuously Generated Data Merge or CGM.

We also introduced an algorithm in which the remote data servers support range queries. Traditionally most of the intelligence for query planning for continuous query systems has been on the client side. For the data grid and data web applications which are our concern, data is served from a server which can support range queries. With this foundation, data clients can use a merge algorithm we call the R-tree merge algorithm or RTM.

In this paper, we show that both algorithms can scale sufficiently to

support remote data analysis and distributed data mining even for high performance OC-12 networks.

We have built prototypes of these algorithms into our open source data servers which provide the foundation for a data web implementation called DataSpace.

References

[1] Protocols and Services for Distributed Data-Intensive Science. A. Chervenak, I. Foster, C. Kesselman, S. Tuecke. ACAT2000 Proceedings, pp. 161-163, 2000.

[2] D. DeWitt, J. Naughton, and D. Schneider, An Evaluation of Non-Equijoin Algorithms, VLDB, Barcelona, Spain, 1991.

[3] J. Chen, D. DeWitt, F. Tian, and Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases, ACM SIGMOD, 2000.

[4] P. Domingos and G. Hulten, Mining High-Speed Data Streams, to appear.

[5] R. L. Grossman, S. Kasif, D. Mon, A. Ramu and B. Malhi, The Preliminary Design of Papyrus: A System for High Performance, Distributed Data Mining over Clusters, Meta-Clusters and Super-Clusters, Proceedings of the KDD-98 Workshop on Distributed Data Mining, AAAI, to appear.

[6] R. L. Grossman, S. Bailey, A. Ramu, B. Malhi and H. Sivakumar, A. Turinsky, Papyrus: A System for Data Mining over Local and Wide Area Clusters and Super-Clusters, Proceedings of Supercomputing 1999, IEEE.

[7] R. L. Grossman H. Sivakumar, S. Bailey. Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks. Proceedings of Supercomputing 2000, November 2000.

[8] R. L. Grossman M. Mazzucco H. Sivakumar Y. Pan Q. Zhang. Simple Available Bandwidth Utilization Library for High-Speed Wide Area Networks, submitted for publication

[9] Robert Grossman, Emory Creel, Marco Mazzucco, Roy Williams, A DataSpace Infrastructure for Astronomical Data, in R. L. Grossman, C. Kamath, W. Philip Kegelmeye, V. Kumar, and R. Namburu, Data Mining for Scientific and Engineering Applications, Kluwer Academic Publishers, 2001.

[10] R. L. Grossman, Standards and Infrastructures for Data Mining, ACM Special Issue on Data Mining, submitted for publication.

[11] R. L. Grossman and M. Mazzucco, DataSpace - A Data Web for the Exploratory Analysis and Mining of Data, submitted for publication.

[12] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan, Clustering Data Streams, to appear.

[13] Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer Generalized Search Trees for Database Systems (1995) Proc. 21st Int. Conf. Very Large Data Bases, VLDB

[14] Phil Lapsley, Network News Transfer Protocol, February 1986.

[15] L. Liu, C. Pu, and W. Tang, Continual Queries For Internet Scale Event-Driven Information Delivery, IEEE Transactions on Knowledge and Data Engineering, 1999.

[16] V. Raman, B. Raman, and J. Hellerstein, Online Dynamic Reordering for Interactive Data Processing In Proc. of the 1999 Intl. Conf. on Very Large Data Bases, 1999

[17] National Center for Atmospheric Research, Community Climate Model. Retrieved from http://www.cgd.ucar.edu/cms/ccm3/ on April 10, 2002.

[18] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani, Streaming-Data Algorithms for High Quality Clustering, to appear.

[19] V. Raman, B. Raman, and J. Hellerstein, Online Dynamic Reordering for Interactive Data Processing In Proc. of the 1999 Intl. Conf. on Very Large Data Bases, 1999

[20] P. Seshadri, M. Livny, and R. Ramakrishnan, The Design and Implementation of a Sequence Database System, VLDB, Mumbai, India, 1996.

[21] D. Terry, D. Goldberg, D. Nichols, and B. Oki, Continuous queries over append-only databases, ACM SIGMOD, pages 321-330, 1992.