A Placement Service for Data Mining Centers

Gregory Buehrer
The Ohio State University
Columbus, OH USA
buehrer@cse.ohio-state.edu

Srinivasan Parthasarathy
The Ohio State University
Columbus, OH USA
srini@cse.ohio-state.edu

ABSTRACT

In this article we describe a placement service for distributed data mining centers. We seek to provide a service with sufficient generality to suit a variety of web mining and machine learning algorithms, yet one which offers properties beyond typical robustness, lookup, and extraction support for general applications. To accomplish this, we design the service to incorporate locality of placement based on the similarity of records in the input data. Specifically, we propose to leverage hashing techniques with low compute complexities to form small, highly similar groups of records which can then be assigned to a machine in the cluster based on the data currently residing on that machine. The subsequent increase in local neighborhoods for the resulting machine partitions can improve mining executions for pattern discovery, nearest neighbor searching, and efficient graph computations, among other applications. We provide the details of the programmer interface and offer an empirical evaluation of the proposed placement techniques. As an examples, we maintain over 40% of the local neighborhood of a large document corpus derived from the web when distributing it over 1024 machines. For large web graphs, we reduce the edge cuts by half when compared to URL ordering over 1024 machines. For fault tolerance, we leverage the local similarity in the partitions to improve indexable compression ratios by over 450%.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining; H.4.m [**Information Systems**]: Miscellaneous

General Terms

Locality, Placement

Keywords

Data Mining Service, Data Placement

1. INTRODUCTION

Advances in data collection technology have led to the creation of large scale data stores. Examples abound ranging from astronomical observational data to social interaction data, from clinical and medical imaging data to protein-protein interaction data, and from the multi-modal content on the World Wide Web to business and financial data routinely collected by enterprises. A fundamental challenge when working with very large data is the need to process, manage and eventually analyze it *efficiently*. A commodity cluster of nodes with a low-latency, high-bandwidth interconnect and with high-capacity commodity disks offers a cost-effective solution to this challenge. However, leveraging the features of such clusters or data centers to deliver end-to-end application performance is still fraught with difficulties.

Three of the most significant issues when employing distributed clusters for data mining are i) localizing computation (maximizing independent computation and minimizing communication), ii) generating a balanced workload, and iii) the need to be fault tolerant (data loss, hardware failures). The placement of data onto a tightly coupled system area network can have a significant impact on the performance of the application in all three of the challenges above, and is therefore the target of this article.

There are several orthogonal dimensions to consider when developing data placement policies. For example, a placement policy might attempt to place closely related pieces of data together to facilitate localized computation. Along another dimension, a policy might favor redundant data placement to facilitate fault tolerant computing. This replication may also be used to improve computation runtimes. A third dimension is disk balance. Support for flexible placement policies is thus highly desirable. In this article we describe the design, development and evaluation of such a flexible placement service for next generation data analysis centers. While the placement service is fairly general-purpose, in this article we place emphasis graph and network structured datasets. Such datasets are fairly ubiquitous, and processing and analyzing them is particularly challenging. For example, computing *betweenness* on the web graph, which is conservatively estimated to be on the order of 20 billion nodes and 500 billion edges, requires a combination of distributed computation, distributed placement, and clever compression techniques.

The central features of our proposed placement framework include

- low-complexity policies to improve similarity between records on a machine,
- efficient, indexable data redundancy
- a flexible hashing mechanism for placement which al-

WWW2008, April 21–25, 2008, Beijing, China.
.

lows user-specified similarity metrics

- a simple-to-use programming interface which supports a variety of data input types

We evaluate our placement framework on several large, real datasets drawn from the web domain. We show the service maintains neighborhoods when moving from a global data structure to the partitioned data for both document data as well as graph data, up to 65% more than typical distribution strategies. Also, it reduces the cuts in a distributed graph from 99% using a round robin scheme to only 7% on 1024 machines. For pattern mining, the local partitions contain over 10-fold more closed patterns at the same support levels. Finally, we illustrate the effectiveness of the placements schemes when compressing the data, reducing the data footprint over 14-fold while still maintaining indices into each record.

2. PRELIMINARIES

In this section we describe the target environment for our service, as well as the existing approaches.

2.1 Model

We develop this placement service for a distributed cluster of PCs executing a variety of data mining applications. The input data is assumed to be in a flat file (two or three files, as described later) which can be accessed by at least one node in the cluster. The challenge is to distribute the data to local disks in the cluster such that a) the load is relatively balanced (from a bytes per node perspective), while b) applications making use of the data have lower runtimes than simple uniform random placement provides. In addition, we would like to incorporate robustness in the placement so that if one node in the cluster is unavailable, that the data can be rediscovered with minimal processing time, automatically in the best case. We assume the existence of a low-level API to address communication between nodes. In our implementation, this interface is MPI.

2.2 Related Research

Distributed data placement has been studied widely in a variety of domains. The peer-to-peer community has studied the challenge extensively. This work usually seeks to afford low costs for discovering the location of a data source, as well as high availability of the data source in the presence of node failures [24, 26, 30]. These works offer improved reliability and discovery over the service we describe in this article, but do not attempt to lower compute times of the using application via localized placement of records within the data sources. For example, *CHORD* is a distributed hash table for peer-to-peer networks. *CHORD* arranges peers in a unit circle, using consistent uniform hashing. It is scalable, decentralized, and highly fault tolerant. Most operations have $O(\log\ n)$ complexity, such as resource discovery.

The grid computing community has studied data placement in detail [2, 27, 20, 23]. This community seeks to estimate the cost of moving computation to the data source, or vice versa, and to provide general tools for reliable data placement. They typically do not attempt to partition the data such that applications leverage which records in the data reside on which nodes in the cluster. For example, *GridFTP*[2] is a mechanism by which grid applications can transfer data reliably and quickly. It has been shown to be highly scalable when using a specialized FTP server, and is provided for download as part of the Globus toolkit. It is a general purpose transfer tool, and does not aid in making locality-specific decisions for applications. A key property of these works is that they target general applicability of the placement services, so they have utility for almost any application, including domains outside data mining. In addition, these works must manage cross-cluster communication and security, challenges we have not addressed here. Our target use case is a tightly connected cluster of PCs residing under a single security.

Many researchers have leveraged clusters of workstations [8] for data mining. Zaki provided a survey of the work [29]. Agrawal and Shafer were the first to address the challenge of frequent itemset mining on shared-nothing architectures, proposing Count Distribution (*CD*) and Data Distribution *DD*[1]. Both *DD* and *CD* are based on *Apriori*. *Decision Miner* [25] was proposed by Schuster and Wolff. It reduces the communication costs of *CD* by pruning candidates which cannot be frequent. The work is similar to *FDM*, but also provides theoretical bounds for the amount of communication reduction. As does *FDM*, *Decision Miner* makes the same number of full data set scans as *Apriori*. Zaki, Parthasarathy, Ogihara and Li parallelized Eclat for distributed memory systems [28]. They place TID lists such that communication is minimized. In all these works, the placement of data is particular to the exact algorithm proposed.

Several researchers have looked at the problem of systems support for distributed data mining algorithms. JAM[22], BODHI[18] and Info-sleuth[19] are high level agent-based distributed systems (in user-space) that support such algorithms. Bailey, Grossman, Sivakumar, and Turinsky developed a data mining framework called Papyrus[14] Papyrus is a layered system of services for distributed data mining and data intensive scientific applications. They define three key data movements, namely move data (MD) move models (MM), and move results (MR). These three operations facilitate interaction amongst nodes in a cluster. The paper presents results for *C4.5*, a common decision tree classifier. The system is designed to allow large data sets to be approximated with a model, so that data transfer is less expensive (the model being much smaller than the data). The tradeoff is a loss of information. The Kensington[7], FreeRIDE[16], and Intelliminer[21] systems look at the problem in terms of clients, and compute and storage servers and implemented basic system level services for data transfer and scheduling of parallel tasks. Jin and Agrawal has several works targeted at solving parallel data mining workloads [16, 17]. They implement a framework for fast prototyping of data mining workloads on shared memory systems. For data placement, they use the general-purpose ADR[1] framework. These works do not preallocate records to generate improved locality for a variety of workloads.

Several works use min-hashing to process data efficiently. Minimum hashing was first proposed by Cohen [9] to estimate the size of transitive closure and reachability sets. The technique was generalized to k-way minimum hashing by Broder and Charikar [4]. The algorithm afforded by Cohen *et al* [10] leverages the k-way hashing technique to discover association rules with high confidence. The authors seek to

[1]http://www.cs.umd.edu/projects/hpsl/ResearchAreas/ADR.htm

find interesting implications ($A \rightarrow B$) with very low support, where A and B are both singletons. This restriction allows for column-wise comparisons for rule detection, but only discovers itemsets of length two. In addition to these smaller patterns, we seek to find very long patterns in the data set. Interestingly, because they are seeking to find two similar items, their matrix is hashed orthogonally to ours. Finally, the authors note that increasing their algorithm to more than 3 columns (and hence patterns of length 4 or more) would suffer exponential overheads. However, their techniques clearly exhibit the general scalability of minimum k-way hashing. Indyk and Motwani [13] used hashing to find nearest neighbors in high dimensional data, developing the LHS technique. Goinis, Indyk and Motwani [15] subsequently improved the idea. They use hashes as signatures to localize data for answering queries in databases. They illustrate the benefits of the technique over sphere/rectangle trees when the number of dimensions is high. Gibson, Kumar and Tomkins [11] use min-hashing to generate large communities in web graphs. They implement a $H(c, s)$ shingling mechanism which effectively uses smaller shingles in the min hash matrix to construct higher level signatures. The work does not find itemsets.

3. PLACEMENT ALGORITHMS

We now describe the details of the proposed placement algorithms. Generally the goal is to group similar nodes together, where similarity is a function of the intersection between records. However, as noted below, the hashing mechanism is designed to allow any user-specified measure.

3.1 Hashed-based Placement

Hashed-based placement uses a hash function to produce a signature for each record in the data set. The signature is typically a fixed width of size K[2]. Users may provide a hashing function, or use one of the several described below. We hash each of the M records of the input data K times to produce an $M * K$ matrix. We then sort M lexicographically. This sort is typically quite fast since M is designed to fit in RAM[3]. Next we traverse the matrix column-wise, grouping rows with the same value. When the total number of rows drops below a user-provided threshold, or we reach the end of the hash matrix, we box the record Ids as a partition of the data, and assign it to a machine in the cluster. For example, suppose in the first column there is a contiguous block of 200,000 rows with the same hash value. We then compare the second column hashes of these 200,000 rows. For each distinct hash value in the second column, we inspect the number of rows with that value. If cardinality is below a user-defined threshold, we pass it to a *partition assigner*; otherwise we inspect the third column hash values for the selected rows, and so on. The lexicographic sort biases the sampling left-wise in the matrix, but affords log-linear complexity. Pseudo code is provided as Algorithm 1.

[2]In practice we use values between 8-32, noting that larger values limit subsequent bias.

[3]If it doesn't, we make multiple Ms that do, performing the partitioning process on each M

Algorithm 1 Hash

Input: Data set D
Input: Hash family $h()$
Input: Number of hashes K
Output: A partition $P = \{p_1, p_2 \dots p_N\}$

```
 1: Matrix M
 2: for all rec ∈ D do
 3:    vector v
 4:    for all k ∈ K do
 5:       Hash h = h(rec, k)
 6:       v_k = h
 7:    end for
 8:    M_rec = v
 9: end for
10: Sort M Lexicographically
11: start = 0
12: end = 0
13: col = 0;
14: currentHash = HashTable[end][col]
15: while end ≤ |D| do
16:    while currentHash == HashTable[end][col] do
17:       end++
18:    end while
19:    GetList(start, end, col + 1)
20:    start = end
21: end while
```

Here we assume the input is a data set of sets – we describe data representations in Section 4.1. Lines 2-9 build the $M * K$ matrix, which is then sorted in line 10. Lines 15-21 generate lists of consecutive indices in the matrix with the same hash value, which are then processed by Algorithm 2. at line 19.

In Algorithm 2, line 1 checks the size of the partition. If the list is large, lines 5-10 scan the next column in the matrix for equal hash values. For each distinct hash value, it generates a recursive call. However, if the index range is below a threshold, a call to a partitioning algorithm is made. Several assignment algorithms are available, depending on the data type (see Section 4.1. We describe mechanisms for graphs and flat records provided as Algorithms 3 and 4.

Algorithm 2 GetList

Input: StartIndex *st*
Input: EndIndex *end*
Input: Index *index*

```
 1: if end − start < THRESHOLD or index = K − 1 then
 2:    AssignPartition(st,end)
 3:    return
 4: end if
 5: currentHash = HashTable[st][index]
 6: while st < end do
 7:    while currentHash==HashTable[st][index] do
 8:       st++
 9:    end while
10:    GetList(st, end, index + 1)
11: end while
```

The *AssignPartitionForGraphs* algorithm provided in Algorithm 3 chooses an appropriate machine for a set of records. In lines 2-13 a histogram is built of the locations of the items of each record passed. Note that these items in the record

correspond to outlinks in the graph. Thus it is desired to locate the machine with the greatest number of outinks, and assign the passed records to that machine. Line 4 dereferences from the hash table index to a record in the data set. Full machines cannot contribute to the histogram. List P retains the record Ids, and the *loc* array stores the assignment of each record in the data set. Lines 15-21 send the records to the machine with the most weight in the histogram and update the weights of that machine.

The *AssignPartitionForTrans* algorithm distributes small groups of items based on a histogram of the elements on each machine. Although seemingly similar to the previous algorithm, we do not dereference items in the records to discover their location, because the items are not assumed to be references to other records. Instead, we maintain a running multiset of each machine's current records. We use a heuristic extension of a well-known algorithm in statistics, which selects any element from a stream of elements with equal probability by selecting the first element and then for every subsequent element, replace the existing element with probability $1/N$, where N is the number of elements seen thus far. We generate our multiset by maintaining a buffer of up to W elements. When the buffer reaches size W, we remove elements with $1/2$ probability. Then, the next record inserted onto the machine updates adds its elements to the multiset with $1/2$ probability. Subsequent halving of the multiset lower the probability of a record adding its elements to the multiset by $P/2$. This set is then used as a signature for the machine. A set of records' distance to the machine is defined as the intersection between the union of the items in the records and the machine's signature set.

Algorithm 3 AssignPartitionForGraphs

Input: startIndex *st*
Input: endIndex *end*
1: List P
2: Histogram h
3: **while** $st < end$ **do**
4: RecordId t =*HashTable[st].recId*
5: **for each** $e \in t$ **do**
6: Machine $m = loc[e]$
7: **if** $m > -1$ and *weight[m]*$<MAX$ **then**
8: $h = h \cup e$
9: **end if**
10: **end for**
11: $p = p \cup t$
12: $st++$
13: **end while**
14: Machine $m = Max(h)$
15: **for each** $t \in p$ **do**
16: Send $Tran[t]$ to m
17: $loc[t] = m$
18: $weight[m] += |t|$
19: **end for**

We briefly review the pseudo code in Algorithm 4. Lines 4-10 build a set S of elements of the records to be assigned. Lines 13-19 find the machine with the largest intersection with S. *Sig* is an array of the multiset signatures. Lines 20-24 send the records to the proper machine and update the proper signature by calling Algorithm 5, *UpdateSignature*. In line 1, this algorithm first calculates the possibility of adding elements to the multiset by using $Prob[m]$, which essentially stores the number of times the signature has been divided. Lines 2-9 perform the (approximate) window halving. Lines 10-12 then add the elements

3.1.1 Min Hashing

The framework for hash-based distribution allows for any user-defined hash function, such as Cosine, Jackknife, etc. We provide two predefined functions. The first is min-hashing[4], uses probabilistic sampling on the data set. Min-wise K-way hashing is a process by which a pseudo random element is chosen from a set with consistency. One uses K independent permutation functions to select k elements from the set. Each permutation function places a different order on the elements in the set, and the first element is considered the signature of that set for that permutation. Performing k permutations arrives at a signature of length k. It has been shown elsewhere [4] that min-hashing approaches the Jaccard coefficient, given below.

$$j = \frac{A \cap B}{A \cup B} \tag{1}$$

Thus if two elements have many similar hash values, then the probability that they have many items in common is high. For many data mining applications, such as graph clustering, etc, this similarity measure is desired.

Algorithm 4 AssignPartitionForTrans

Input: StartIndex *st*
Input: EndIndex *end*
Input: Histogram *centers*[]
1: List P
2: Histogram h
3: Set S
4: **while** $st < end$ **do**
5: RecId t = *HashTable[st].recId*
6: $P = P \cup t$
7: **for each** $e \in t$ **do**
8: $S = S \cup e$
9: **end for**
10: **end while**
11: $maxid = -1$
12: $max = -1$
13: **for each** $m \in MS$ s.t. *weight[m]*$<MAX$ **do**
14: $prox = Intersection(S, Sig[m])$
15: **if** $prox > max$ **then**
16: $max = prox$
17: $maxid = m$
18: **end if**
19: **end for**
20: **for each** $t \cup P$ **do**
21: Send $Tran[t]$ to m
22: $weight[m] += |t|$
23: *UpdateSignature(t,m)*
24: **end for**

3.1.2 Approximate Hashing

Approximate hashing provides a mechanism to partition data such that a prefix tree can be constructed in multiple parts independently [6]. The idea is to sample the data to create a histogram, then on a subsequent full pass use a geometric binning procedure to move each transaction into a particular bin. All the items in bin b_i sort before items in b_{i+1}. If the data is highly skewed, which can be obtained

from the initial scan, then each item in the sorted histogram receives twice the storage as the following bin.

Since transactions typically contain many items, the second item in a transaction partitions the sub-bins as it partitioned the original bin space. For example, suppose items a and b are the two most frequent items in the data set. If bin $b_1 \dots b_{10}$ were assigned to item a and $b_1 1 \dots b_1 5$ were assigned to item b, then bins $b_1 \dots b_5$ would contain transactions having both *a and b*. We generalize the procedure to fit our hashing model above. For approximate sorting, the *kth* hash function selects the *kth* most popular item for the hashed transaction. The algorithm runs in linear time in the size of the data set and log-linear in the size of the sample.

Algorithm 5 UpdateSignature

Input: Machine m
Input: RecId t
1: **if** $rand()\%Prob[m] = 0$ **then**
2: **if** $|Sig[m]| > W$ **then**
3: **for each** $w \in Sig[m]$ **do**
4: **if** $rand()\%2 = 0$ **then**
5: $Sig[m] = Sig[m] \setminus w$
6: **end if**
7: **end for**
8: $Prob[m]* = 2$
9: **end if**
10: **for each** $e \in t$ **do**
11: $Sig[m] = Sig[m] \cup e$
12: **end for**
13: **end if**

3.2 Other Placement Routines

Sorted The interface provides for an arbitrary sorting routine to be used. The mechanism is to then sort the data per the passed function, and partition the data evenly based on the total number of bytes assigned to each machine (subject to record boundaries).

Round Robin It may be the case that the user desires a relatively uniform distribution of data on each machine. The round robin scheme is advantageous in this case, because unlike randomly choosing a machine for each cluster, the pattern is regular. Therefore, no index file is needed, and each node can calculate at runtime the machine mapping for every record.

3.3 Redundancy

We address fault tolerance by adding a replicated but compressed form of each partition. The default placement of the compressed partition is the next machine Id in the mapped cluster. For example, define the partitioned data as

$$P = \{p_1, p_2, p_3 \dots, p_N\} \tag{2}$$

where i is the machine id for subset p_i. Then the compressed data is defined as

$$C = \{c_1 = f(p_N), c_2 = f(p_1), \dots c_N = f(p_{N-1})\} \tag{3}$$

where $f(p)$ represents the compressed form of p.

Two compression schemes are provided. The first uses approximate frequent itemsets in records to locate redundancy, and then compresses these itemsets into a pointer to the set. By maintaining only one copy of the itemset, all but one occurrence of the pattern can be stored using one 4 byte value (if there are less than 4 billion records). The algorithm is efficient when compared to traditional itemset mining – it runs in $O(E \log E)$ compute complexity. In addition, it affords compression ratios up to 15-fold for many large data sets. Further details can be found elsewhere [5]. Two caveats are that i) the mechanism is dependent on correlation of items to be effective, and ii) the order of elements within a record is not respected. This latter issue acceptable for adjacency lists and transactional data but will offend input data where order implies information, such as bitwise feature vectors. In addition to affording improved compression rates over other methods, it also allows direct indexing into the records in compressed form, since all compression patterns exist within the boundaries of a record. Thus, for space conscious users, this format can serve as both the main data file as well as the backup file.

The second compression scheme is to compress the partitions using traditional techniques such as *LZ77*. The placement location of the compressed chunk is the same as the previous method. *LZ77* often provides excellent compression ratios for a variety of data. The main drawback is that the compressed file must be fully uncompressed to be used.

4. PLACEMENT SERVICE API

In this section we present the service interface to the programmer. A high-level overview is provided as Algorithm 6.

4.1 Data Representation

The user provides input data as transactions (or records) using two or three files. The first input file is a binary string of the elements in the records. The second input file is an index which stores offsets for each record. For example, the *ith* value in the offset file is the starting address of the ith record in the record file. The length of the *ith* record is simply $index[i+1] - index[i]$. The last record uses the length of the record file as the secondary boundary. The optional third file is an XML meta data file. Each record in this third file is of the form *<Rec>text</rec>*. Child nodes of *<Rec>* may be added, as they simply pass through our representation unaltered. We now describe how the first two files are used to represent the many input formats of typical data mining stores.

One large graph or tree can leverage the above input format via an adjacency list, where each index is a consecutive node in the graph. Urls and other meta data are stored in the meta file. A set of graphs uses the index file to position each graph in the record file. The record file first lists V values representing node labels, then a '-1', followed by E tuples in the form of *SourceNode,DestinationNode,EdgeLabel*. A set of trees follows the same format. If an index file only has one value, then it is assumed that each record has length of that value.

Currently 6 data types are available. These include 1 byte, 2 byte, 4 byte, 8 byte, 4 byte float and 8 byte floats. We require this size for the elements in a record for hashing purposes. Again, any ASCII data is stored in the meta file and is not hashed.

4.2 Bootstrapping

The service is started with by calling $Ss :: StartService()$. The save path and input data filename must first be provided. Currently, it is assumed that the index file can be obtained by appending '_index' and the meta data file can be obtained by appending '_meta'. Each registered data input is maintained in the user's *Manifest* file. The manifest file is an XML file containing the pertinent parameters necessary to verify the full data set is present. It includes the filename, the save path used, the number and names of the machines used for the original partitioning, the distribution type, the data type, the local data sizes and the index type.

Algorithm 6 Placement Service Programmer Interface

```
1: bool StartService(int rank,int nTasks);
2: bool ShutDown();
3: void PrintManifest();
4: void ResetManifest();
5: bool SetFilename(std::string filename);
6: bool SetSavePath(std::string filename);
7: bool AddData(int indexMode, int distMode, int dataSize, int dataType);
8: bool AddData(int indexMode, int distMode, int dataSize, int dataType, void* Comparator);
9: bool ReplicateData(bool UseHashCompression);
10: bool DeleteData();
11: bool DataExistsInAnyManifest(std::string filename);
12: bool RetrieveData(File* data, File* index);
13: bool RetrieveData(File* data, File* index, File* metaData);
14: bool VerifyData();
15: bool VerifyDataFilesExist();
16: bool VerifyNumberOfServers();
17: enum {Transactions,Graphs,Trees,OneGraph,OneTree};
18: enum {Range,LocalItems,GlobalKnowledge};
19: enum {SingleMachine, Random, RoundRobin, MinHash, Hashed,ApproxHash, Sorted, Euclidean, Serial};
20: enum {1byte,2byte,4byte,8byte,4byteF,8byteF};
```

4.3 Adding Data

Users add data by calling one of the two *AddData()* functions on lines 7 and 8. They provide a distribution mode (line 19), an index mode (line 18), a data size (line 20), a data type (line 17), and possibly a function pointer.

Distribution Modes There are eight distribution modes, which are specified during the function call using the enumeration on line 19. *Single Machine* stores all the records on a single machine. *Random* places records randomly on the cluster. It requires either the *Local* or *Global* index types, as ranges are not practical. *Round robin* places record r on machine r modulo M. It does not require an index. *Min hashing* uses the distribution algorithm outlined in the previous section, as does *Approximate sorting*. *Hashed* allows for a custom hash function family. Data added in this manner must use the function on line 1, and pass the hashing function family as function pointer which accepts a record id and K and returns a pointer to the hashes. *Serial* partitions the data evenly across the cluster starting with the first $|D|/M$ records being placed on machine 0. *Sorted* does not hash, but sorts the data directly, based on the comparator passed. This comparator may access meta data from the meta data file by accessing the field *service.meta[i]*. *Euclidean* uses standard kMeans to distribute the data set.

Index Modes The index mode determines the amount of information stored at each machine regarding the location of the records. Four index modes are available. *Ranged* provides the ranges of records for each machine. All machines have knowledge of the location of every record. This mode is most efficient for the *Serial* distribution mode. *Local items* provides in the mapping file each local item id, which maps in order to the index file. The user must query the other machines for knowledge of records off the local machine. Lastly, in the *Global knowledge* index mode, each machine has the location of each file in the data set.

Algorithm 7 Example Program Adding Data

```
1: Mpi_Init(&args, &argv);
2: Mpi_Comm_size(Mpi_Comm_World, &nTasks);
3: Mpi_Comm_rank(Mpi_Com_World, &myRank);
4: std::string inputFile(argv[1]);
5: Ss service;
6: service.SetFilename(inputFile);
7: service.SetSavePath("/scratch/buehrer/");
8: service.StartService(myRank,nTasks);
9: service.AddData(Ss::MinHash, Ss::GlobalKnowledge);
10: service.PrintManifest();
11: if myRank==0 then
12:    cout << 'Size=' << nTasks << endl;
13: end if
14: File *data, File *index;
15: bool res = service.RetrieveData(data,index);
16: UInt64 size = GetFileSize(index);
17: for int i=0;i<size;i++ do
18:    vector<int> *rec = GetRec(data,index,i);
19:    Process_Rec(rec);
20: end for
21: service.ShutDown();
22: Mpi_Finalize();
23: return 0;
```

4.4 A Typical Use Case

We provide a simple example to illustrate the utility of the proposed service API, as shown in Algorithm 7. Note that there is currently no install procedure – the user simply includes the C++ header files for the service. Lines 1-3 are typical to MPI programs. In line 6 the programmer provides a pointer to the input data. Line 7 sets the save path for partitioned data. This path must be accessible by each node in the cluster. The example uses local scratch space. Line 8 starts the service. This call opens the local manifest file on each machine, which stores the relevant information for all data sets which have been assigned to each node. Line 9 adds the data set to the cluster. The user chooses a Service state is maintained on a per-user basis. Each data object is stored in an XML file in the user's local scratch space on the cluster [4]. Subsequent calls to process the data use the *Retrieve()* function without adding the data source.

5. EVALUATION

In this section, we evaluate the algorithms empirically. All algorithms were implemented on Linux in C++. In all cases we use 8 hashes; more provided little compression benefit, and incurred a linear cost in execution time. The data

[4]We do not address security issues at this time

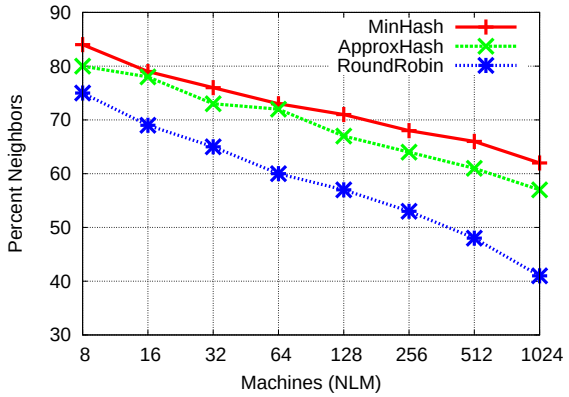

Figure 1: The ability of the placements algorithms to maintain neighborhoods for the *NLM* data set.

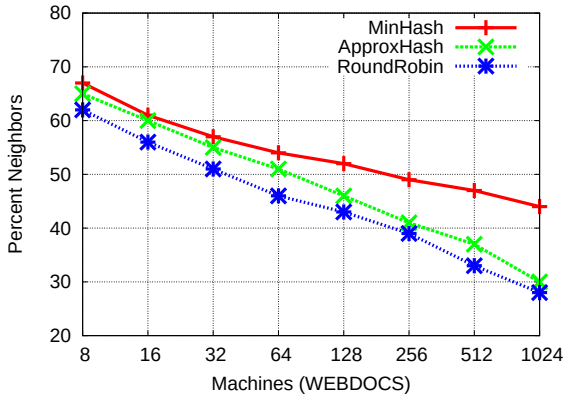

Figure 2: The ability of the placements algorithms to maintain neighborhoods for the *Webdocs* data set.

sets [3] used are presented in Table 1. The web graphs are publicly available[5]. Each is a graphical representation of a portion of the world wide web, where a node is a web page and a hyperlink represents a directed edge. *NLM* is a 2006 snapshot of the National Library of Medicine's MEDLINE/PubMed database (English documents only) of abstracts. There are about 8.5M documents included here, normally distributed in length. Each record is a document, and each item is a word id (stop words have been removed). The *Webdocs* data set was obtained from the *FIMI* Repository[12]. It is a collection of web documents, using word Ids as the items in each record. Most general-purpose placement middlewares will use either i) user-directed placement or ii) a random or round robin placement. Therefore, in this section we will make use of the round robin placement scheme for comparison purposes.

5.1 Maintaining Neighborhoods

We evaluate the placement service to maintain nearest neighbors when distributing the data set. The premise is that if similar documents are partitioned onto the same machine, then local classification will produce more accurate results with nearer neighbors, potentially lowering the need for inter machine communication. First the 20 nearest

[5]http://law.dsi.unimi.it

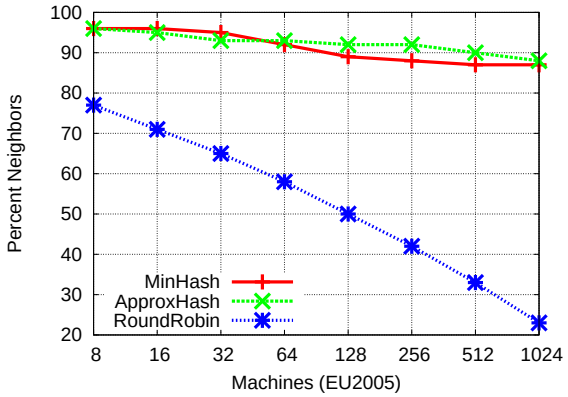

Figure 3: The ability of the placements algorithms to maintain neighborhoods for the *EU2005* data set.

neighbors are found for 1000 random records in the data set. The Jaccard coefficient is used for similarity, and the top 20 scores are summed for each record. These sums are then summed to produce a final value. We then distribute the data, with both *min-hashing* and *round robin*. Again these 1000 records are located, and their 20 nearest neighbors are found – with the constraint that the neighbor must be on the same machine as the record in question. These values are then summed as before. We divide the sum from each partitioning by the global sum to obtain a ratio of *maintained neighbors*. Although straightforward, this experiment captures precisely what we hope to achieve, which is to group similar items onto the same machine, with little overhead.

These values are plotted for various machine cluster sizes operating on the *NLM*, *webdocs* and *EU2005* data sets in Figures 1 2, and 3 respectively. In all cases, local neighbors are better maintained by *min hashing*. For example, with the *webdocs* data set, the ratio drops from 62% at 8 machines to 28% on 1024 machines using the *round robin* scheme. This is understandable, as the neighborhood is essentially evenly distributed, and finding new local neighbors with high similarity is unlikely. However, using min-hashing the ratio only drops from 68% to 44% over the same range of machines. The reason is that many of the local neighbors are assigned to a machine together when they are passed to Algorithm 4. *NLM* exhibits similar behavior, degrading from 84% to 62% using min-hashing whereas round robin degraded from 75% to 41%. The *EU2005* data set has the largest disparity between hashing and round robin, because it is a web graph and the each node has several highly similar neighbors which are found easily by hashing but lost in the round robin scheme. We note the difference between approximate hashing and min-hashing was modest. Also, the load was well distributed, with standard deviations of local sizes below 50% of the average load.

5.2 Partitioning Large Graphs

We also analyze the ability of our hashing to partition a large web graph data on a cluster. The quality metric used is the percentage of graph edges that reside on the local machine divided by the total number of edges in the graph. So a *percent cut* of 25 means that 3 out of every 4 links point to vertices residing on the local machine. We use two graphs, namely *EU2005* and *UK2002*. Figure 4 depicts the results

Data Set	Records	Items	Item/Rec	size	Type
UK2002	18,520,487	298,113,762	16.09	1.2GB	web graph
ARABIC2005	22,744,080	639,999,458	28.13	2.4GB	web graph
EU2005	862,664	19,235,140	22.29	77MB	web graph
NLM abstracts	8,509,325	588,802,208	69.2	2.2GB	documents
Webdocs	1,692,000	397,000,000	234	1.48GB	documents

Table 1: Data sets.

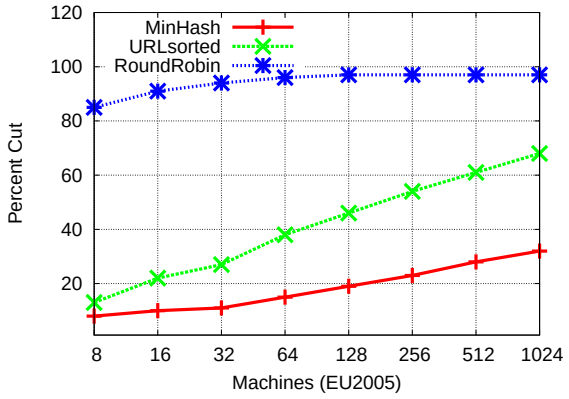

Figure 4: The percent cuts for partitioning the *EU2005* data set.

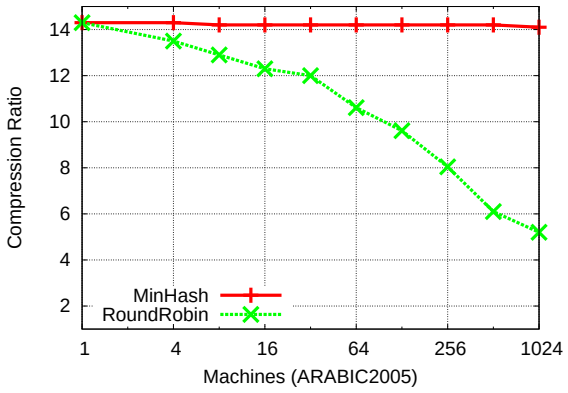

Figure 6: Compression ratios as a function of the cluster size for the *ARABIC2005* data set.

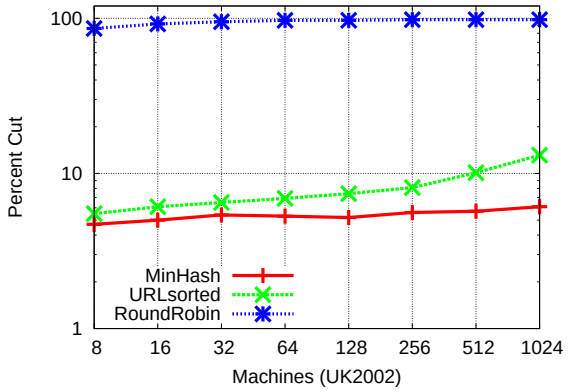

Figure 5: The percent cuts for partitioning the *UK2002* data set.

of partitioning *EU2005*. In this figure, min-hashing is compared to round robin as well as a custom sorting function, which is to sort the graph by the URL. We include URL ordering because it is a popular and effective custom sorting routine for grouping similar nodes in web graphs. As can be seen, hashing results in a lower number of cut edges when compared to the other two methods. For example, hashing incurs less than half the edge cuts for *EU2005* on 1024 machines (32% vs. 68%).

However, unlike the neighborhood study in the previous experiment, it does suffer load imbalance. We inspected this challenge by using four different balance factors and measuring the load distributions. Roughly speaking, the balance factor is the probability that we assign the the records passed to Algorithm 3 to the location with the highest outlink correlation. The graphs in Figure 8 provide the standard deviation (and the average) of the load for a given balance factor, as well as the percent edge cut. It can be seen that the standard deviation can be controlled, but at the cost of increased edge cuts. There is clearly a tradeoff between the balance of the load and the improvement in placement. Still, in all cases the total number of cut edges is vastly lower than what is offered by a round robin placement scheme. Another point to consider is that 1024 machines is a large number of machines to use to partition the smallish (77MB) *EU2005* graph.

5.3 Compression Support

We evaluate our compression technique for its ability to compress the data set as we distribute it across the cluster. The procedure is as follows. We partition the data onto the cluster. Next, we compress the data on each machine, and move this compressed replication to its proper machine. We then sum the sizes of the compressed files and divide the original filesize by this sum to arrive at a compression ratio. We present the results in Figures 6 and 7, for the *ARABIC2005* and *EU2005* data sets, respectively. In the figures, *Machines=1* equates to compressing the full file on one machine. This value will be a compression maximum for the data set, since it can make maximum use of replicated itemsets. We present two curves in each, which are the compression ratios when partitioning with *min-hashing* and with *round robin*.

First let us consider Figure 6. The maximum compression for *ARABIC2005* is a 14.3-fold reduction when compressed on a single machine. As we partition the data, it is clear that *min-hashing* maintains a larger percentage of its locality, degrading only to a ratio of 14.1-fold when distributing over 1024 machines. However, *round robin* degrades to a compression ratio of only 5.3-fold, because it does not keep similar records on the same machine.

Next we consider Figure 7. The maximum compression

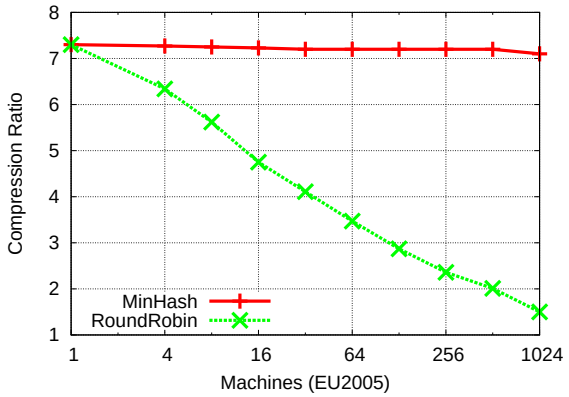

Figure 7: Compression ratios as a function of the cluster size for the *EU2005* data set.

EU2005 is a 7.3-fold reduction when compressed on a single machine. As we partition the data, again *min-hashing* maintains a larger percentage of its locality, degrading only to a ratio of 7.1-fold when distributing over 1024 machines. However, *round robin* degrades to a compression ratio of only 1.53-fold, because it does not keep similar records on the same machine. Hashing offers more than 450% more compression. Some degradation can be attributed to the small size of the input data as well.

We also compressed the input data using *gzip (LZ77)*, which does not provide record level access but may be acceptable to many users. The *ARABIC2005* data set compressed 14.2-fold and the *EU2005* data set compressed 8.6-fold. When partitioned across the cluster, the hashing partitioning schemes provided better *gzip* compression than *round robin*, typically around 1.75-fold better [6].

5.4 Frequent Patterns

In our final set of experiments, we briefly explore the local partitions for frequent itemsets. It is well known that mining for global itemsets in a distributed environment is an expensive operation. Thus, we seek to group similar transactions so that relevant patterns can be discovered locally, while maintaining the same support threshold. The benefit is that local patterns can be mined without communication of meta data or portions of the data set. To evaluate this operation, we first mine the data set at a given support on a single machine. Then, we partition the data set and mine each local partition at the same relative support. If we have increased the relative associativity in the data set, we can discover interesting patterns which would require a much smaller support threshold if mined globally.

We use the *EU2005* data set at a support of 20%. It contained 25 global closed patterns. When we partitioned it using min hashing on 8 machines, using the same relative support, we discovered 3852 closed patterns. When using round robin distribution, we discovered 132 closed patterns. In fact, discovering more patterns through hash-based placement occurred n all cases we tested.

6. CONCLUSION

[6]graphs omitted for space purposes

In this article we engineer an API for data distribution on large data mining centers. The interface and algorithms are sufficiently general so as to support a variety of common data mining workloads, while providing the potential for improved application performance through locality sensitive placement. We demonstrate this potential for several key mining workloads. For classification, we demonstrate our methods maintain better local neighborhoods that other placements schemes, allowing similar records to be placed on the same machine at low cost. Also, we demonstrate the ability to lower inter-machine links when distributing large web graphs. Third, we present a distributed fault tolerant compression mechanism, for large graphs which affords comparable compression to traditional full compression schemes, while still affording node level access in the compressed form. Finally, we are in the process of implementing and testing several full applications using the proposed service and look to present these efforts in future work.

7. REFERENCES

[1] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[2] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The globus striped gridftp framework and server. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 54, Washington, DC, USA, 2005. IEEE Computer Society.

[3] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. In *Software: Practice & Experience*, number 8, pages 711–726, 2004.

[4] Andrei Z. Broder, Moses Charikarand Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *Journal of Computer and System Sciences*, volume 60, pages 630–659, 2000.

[5] Gregory Buehrer and Kumar Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *To appear in the first international conference on web search and data mining*, 2008.

[6] Gregory Buehrer, Srinivasan Parthasarathy, and Amol Ghoting. Out-of-core frequent pattern mining on a commodity pc. In *KDD '06: Proceedings of the 12th ACM SIGKDD international con ference on Knowledge discovery and data mining*, pages 86–95, New York, NY, USA, 2006. ACM Press.

[7] J. Chattratichat, J. Darlington, Y. Guo, S. Hedvall, M. Koller, and J. Syed. An architecture for distributed enterprise data mining. In *HPCN europe, pp.573-582*, 1999.

[8] A. Cheung and A. Reeves. High performance computing on a cluster of workstations. In *Proceedings of the Symposium on High Performance Distributed Computing (HPDC)*, 1992.

[9] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. In *Journal of Computer and System Science*, volume 55, pages 441–453, 1997.

[10] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *IEEE Transactions on Knowledge and Data Engineering*, volume 13, 2001.

[11] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proceedings of 31st International Conference on Very Large Data Bases (VLDB)*, 2005.

[12] B. Goethals and M. Zaki. Advances in frequent itemset mining implementations. In *Proceedings of the ICDM workshop on frequent itemset mining implementations*, 2003.

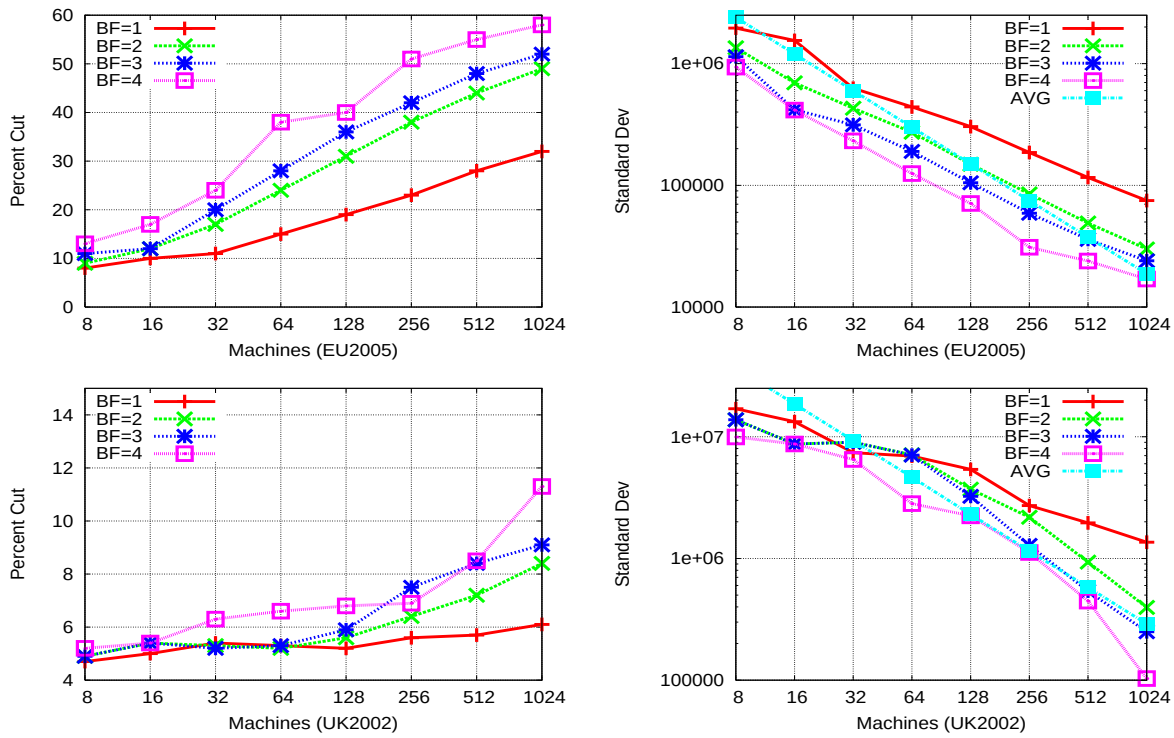

Figure 8: Percent cuts and the load imbalance as a function of the balance factor for the *EU2005* (top) and *UK2002* (bottom) data sets.

[13] A. Goinis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*, 1999.

[14] R. Grossman, S. Bailey, A. Ramu, B. Malhi, and A. Turinsky. The preliminary design of papyrus: A system for high performance, distributed data mining over clusters, meta-clusters and super-clusters. In *Advances in Knowledge Discovery and Data Mining*, 2000.

[15] P. Indyk and R. Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual Symposium on Theory of Computing*, pages 604–613, 1998.

[16] R. Jin and G. Agrawal. A middleware for developing parallel data mining implementations. In *Proceedings of SIAM International Conference on Data Mining (SDM)*, 2001.

[17] R. Jin and G. Agrawal. Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance. In *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.

[18] H. Kargupta, B. Park, D. Hershberger, and E. Johnson. Collective data mining: A new perspective toward distributed data analysis. In *Advances in Distributed and Parallel Knowledge Discovery, Kargupta and Chan ed.*, 2000.

[19] G. Martin, A. Unruh, and S. Urban. An agent infrastructure for knowledge discovery and event detection. In *Microelectronics and Computer Technology Corporation Tech. Rep. MCC-INSL-003-99*, 1999.

[20] Sivaramakrishnan Narayanan, Tahsin Kurc, Umit Catalyurek, and Joel Saltz. Database support for data-driven scientific applications in the grid. In *Parallel Processing Letters*, volume 13, pages 245–271, 2003.

[21] S. Parthasarathy and R. Subramonian. Facilitating data mining on a network of workstations. In *Advances in Distributed and Parallel Knowledge Discovery, Kargupta and Chan ed.*, 2000.

[22] Andreas L. Prodromidis, Philip K. Chan, and Salvatore Stolfo. Meta-learning in distributed data mining systems: Issues and approaches. In *Advances in Knowledge Discovery and Data Mining*, 2000.

[23] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *11th IEEE International Symposium on High Performance Distributed Computing*, 2002.

[24] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenke. A scalable content-addressable network. In *Proceedings ACM SIGCOMM*, 2001.

[25] Assaf Schuster and Ran Wolff. Communication-efficient distributed mining of association rules. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 473–484, New York, NY, USA, 2001. ACM Press.

[26] I. Stoica, R. Morris, D. Karger, F. Kasshoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings ACM SIGCOMM*, 2001.

[27] George Kola Tevfik. Run-time adaptation of grid data placement jobs.

[28] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. In *Scalable High Performance Computing for Knowledge Discovery and Data Mining*, 1998.

[29] Mohammed J. Zaki. Parallel and distributed association mining: A survey. In *IEEE Concurrency, special issue on Parallel Mechanisms for Data Mining*, volume 7.4, pages 14–25, 1999.

[30] Ben Zhao, Ling Huang, Jeremy Stribling, Sean Rhea, Anthony Joseph, and John Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE JOURNAL ON SELECTED AREAS IN*

COMMUNICATIONS, volume 22, pages 41 – 53. IEEE, 2004.