A Sorting Service for Next Generation Data Analysis Centers *

G. Buehrer[1], S. Parthasarathy[12], A. Ghoting[1], Xi Zhang[1], S. Tatikonda[1], T. Kurc[2], and J. Saltz[12]
The Ohio State University, Columbus, OH, USA

Advances in data collection and storage technologies have given rise to large dynamic data stores. In order to effectively manage and mine such stores on modern and emerging architectures, one must consider both designing effective middleware support and re-architecting algorithms, to derive performance that commensurates with technological advances. In this report, we present a sorting algorithm for preplacing transactions on disk, designed to support middlewares for data mining.

1 Introduction

For many large data sets, efficient placement on disk is a critical step towards an effective solution. We build into our framework a sorting service designed to partition large data sets across the cluster. This service is designed to accommodate dynamic updates to the data stores, and data input is assumed to be streaming. Each node can accept new data. Data sets are partitioned into datalets, which are data blocks bounded by the main memory of the host node. This bound accommodates efficient in-memory sorting when needed. When an incoming data object is processed, the receiving node checks its manifest for the mapping between the node and the object. The object is then transferred to the appropriate host. In some cases, the datalet may reach its maximum size. At that time, the block is redistributed across the cluster, and a global communication proceeds to update all manifests.

The particular comparison function for sorting is datalet driven. For transactional data, we implement two datalet manifests. The first method sorts transactions based on its items. The second method requires the datalet at the incoming node to maintain a frequency for each unique item in the data set, and sorts the transaction based on frequencies, where the most frequent item is sorted first. This second method requires additional information to be propagated by nodes which receive the original transaction, but it allows for highly efficient mining. Both methods begin with a log distribution, and use frequency data to reform blocks to linear distributions based on parity.

```
Input: Transaction T, int totalFiles
Output: int X_t

CalculateFileNo(Transaction T,int totalFiles)
(1) min = 0
(2) max = totalFiles
(3) check = 0
(4) index = 0
(5) While (min < max and index < |T|)
(6)     If (T[index] == check)
(7)         max = (min+max)/2
(8)         index++
(9)     Else
(10)        min = (min+max+1)/2
(11)    check++
(12)End While
(13)Return min
```

Figure 1. Geometric Partitioning Algorithm.

2 Sorting Service Overview

When executions force meta structures to spill onto disk, many algorithms exhibit severe performance degradation. This is understandable; typical main memory access times are about 15 nanoseconds, while typical disk access times are 5 milliseconds, constituting a 333,000-fold gap in performance. This gap is likely to widen in the future because memory access times are improving faster than disk access times. Our strategy to achieve an out-of-core solution is to minimize the performance degradation due to this gap through data and computation restructuring, to improve locality. In this Section, we present a sorting technique for improving the I/O performance of frequent itemset mining algorithms. We choose to present this technique via *FPGrowth* because it has been shown to be the most efficient frequent pattern mining algorithm to date [1].

*This work is primarily supported by NSF grant #NGS-CNS-0406386. The authors would also like to acknowledge NSF grants #CAREER-IIS-0347662 and #RI-CNS-0403342. [1]Department of Computer Science and Engineering, [2] Department of Biomedical Informatics. Contact email: srini@cse.ohio-state.edu

Input: int N, partition X
Output: int X_t

CalculateFileNo(Transaction T,int totalFiles, int index, int check)
(1) If ($|X| > 2N$)
(2) If (T[index] $<$ check+N
(3) X_t = T[index] - check
(4) Else
(5) div = (totalFreqItems - check)/($|X| - N$)
(6) X_t = (T[index] - check)/div
(7) Else
(8) div = (totalFreqItems - check)/($|X| - N$)
(9) X_t=(T[index] - check)/div
(10)Return X_t

Figure 2. Arithmetic Partitioning Algorithm.

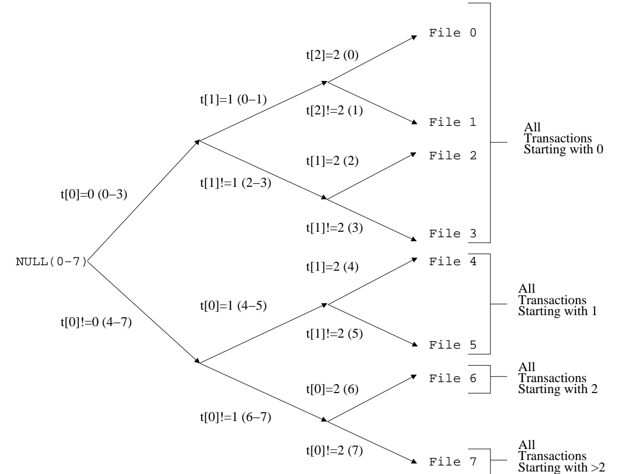

Figure 3. Geometric Partitioning Decision Diagram, for 8 Files.

2.1 Approximate Hash Sorting

As discussed earlier, the initial step in *FPGrowth* is to construct a global prefix tree. This first tree can be quite large; at low supports it can approach or even exceed the size of the data set. For in-core data sets, this construction time is typically a small percentage ($<$ 5 %) of the total mining time. For out-of-core data sets, however, construction of the first tree results in severe performance degradation. During an empirical study we found that our cache-conscious algorithm was a significant performance improvement over *FPGrowth* for out-of-core data sets. However, it spent over 90% of the execution time building the first tree, due to an excessive number of page faults. The reason is that transactions within the data set appear randomly, which results in random writes to the tree nodes in virtual memory during tree construction. Even if the initial data set had its transactions ordered, the problem would persist since the transactions are relabeled prior to tree construction (for improved overlap).

Our solution to this problem is to redistribute and approximately sort the transactions after the first scan of the database. Naturally, sorting on disk is quite slow. Traditional methods for external sorting (such as B-tree insertion and disk-based merge sort) do not provide an overall performance improvement. Exact sorting requires too much time. Instead, we leverage domain knowledge and the frequency information collected in the first scan to approximately sort the frequent transactions into a partition of blocks. Each block is implemented as a separate file on disk. The algorithm guarantees that each transaction in block_i sorts before all transactions in block_{i+1}, and the maximum size of a block is no larger than a preset threshold. By blocking the frequent data set, we can build the tree on disk in fixed memory chunks. A block as well as the portion of the tree being updated by the block will fit in main memory during tree construction, reducing page faults considerably.

We use frequency distributions to choose one of the two partitioning algorithms listed in Figures 1 and 2 by building a simple model of the distribution. We build this model using the top 10% most frequent items. Essentially, if the item frequencies follow a geometric series (in descending order), we partition based on the algorithm in Figure 1, otherwise we partition based on the algorithm in Figure 2.

We first describe the algorithm in Figure 1. Let X= $|partition|$, or the total number of files. Let S represent the maximum file size. We define a function such that transactions with the most frequent item receive the top X/2 of the blocks, transactions with the second most frequent item receive the next X/4 of the blocks, and so on. Of these top X/2 partitions, the top X/4 are dedicated to the subset which also contain the second most frequent item. The bottom X/4 blocks of this subset are split into two equal sections. The top X/8 is dedicated to transactions containing the third most frequent item exists, and the lower X/8 for those which do not. This pattern recurses until the exact block number is known. Therefore, *in one scan*, each transaction is inserted into one of X blocks (typically 256), based on its contents. In the case that a partition has a size above our threshold, we evaluate its local distribution and process it recursively.

Let us illustrate this algorithm with a simple example. Suppose transaction T = {33, 11208, 11, 678, 14, 91, 278}. After scanning the data set, calculating frequencies (removing infrequent items) and relabeling, the transaction becomes T' = {0, 1, 4, 6, 10}. Let the number of files $|X|$ = 8, as in Figure 3. Our task is to de-

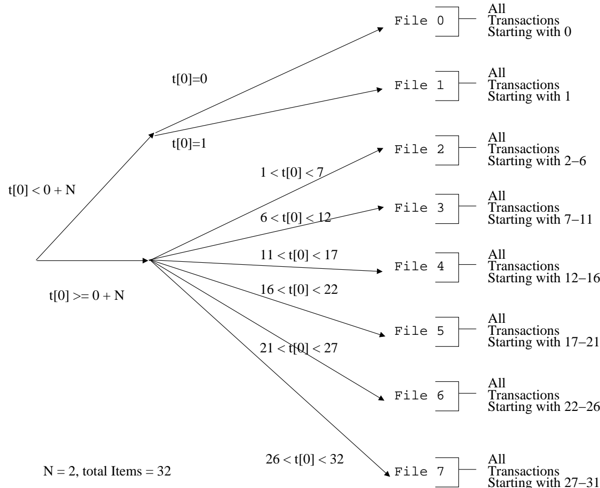

Figure 4. Arithmetic Partitioning Decision Diagram, for 8 Files.

termine X_t, the file assigned to T'. We examine the first element in the transaction, and if it is the smallest (most frequent) element possible, we assign it to the upper half of the possible files. In our example, $0 = 0$ (first item in transactionis the most frequent item), so we reduce the potential file assignment to 0-3. The second element is also its minimum (1=1,second item is the second most frequent item), and therefore the file list is reduced to 0-1. The third element is not its minimum ($4 \neq 2$, or the third item in the transaction is not the third most frequent item), so the block is assigned to file X_1.

If the item frequency model more closely resembles a linear distribution, we partition using the algorithm in Figure 2. Effectively, we assign the first N blocks to the most frequent items, and assign the remaining items to the remaining files equally. Lines 2−6 assign the transaction to either a *dedicated* file if the item in the index is highly frequent, or a *shared* file if the item is not highly frequent. High frequency is relative; we allow for a parameter N to distinguish the threshold for the top items which receive dedicated files. In practice we set N to 5% of the total number of frequent items. A decision diagram for this algorithm is presented in Figure 4. As an optimization, if two consecutive items are both highly frequent, then we skip their indices when sorting, since the count information does not help to partition the data.

As stated earlier, it may be the case that a resulting file in the partition exceeds our threshold S. If so, we simply recurse on the file with the same procedure. However, we must calculate the new start index (in the transaction) to continue the partitioning. The index can be determined solely based on the file number, using the algorithm provided in Figure 5. Note that this results in $n * |partition| - (n - 1)$ files in total, where n is the number of file splitting calls. For sub-splitting files which

---

Input: int N, partition X
Output: int index

CalculateFileNo(int fileNo)
(1) index=0
(2) x = $log_2|X|$
(3) check = x
(4) For $(i = 1; i < x; i++)$
(5) If (fileNo mod $2^i < 2^{i-1}$)
(6) index++
(7) Else
(8) Break
(7) Return index

Figure 5. Recursive Index Calculation.

surpass the maximum file size, we may neglect to build a model of the distribution for that subfile. Note that odd file numbers contain transactions whose last element was not the minimum (most frequent) value possible, and even valued file numbers contain transactions whose last element was a minimum. In practice we have found file parity provides sufficient information to evaluate which algorithm to use when sub-splitting; even numbered files are partitioned geometrically and odd numbered files are partitioned arithmetically.

With the knowledge that consecutive files are in relative order, tree building can proceed by processing the files in order with a minimum number of page faults. This partitioning technique dramatically reduces the cost of building the main tree on disk, and provides a significant improvement to the total execution time.

References

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