5th International Workshop on High Performance Data Mining: Resource and Location Aware Mining (HPDM:RLM'02)

Organizers:

Srinivasan Parthasarathy,

Hillol Kargupta,

Vipin Kumar,

David Skillicorn,

Mohammed Zaki

Date: April 13 2002

International Programme Committee

E. Bertino, DSI, University of Milan, Italy
D. Cheung, University of Hong Kong, Hong Kong
A. Choudhary, Northwestern University, USA
Alex A. Freitas, PUC-PR (Pontifical Catholic University of

Parana), Brazil
J. Gehrke, Cornell University, USA
R. Grossman, University of Illinois-Chicago, USA
Y. Guo, Imperial College, UK
H. Kargupta, University of Maryland (BCC), USA
V. Kumar, University of Minnesota, USA
Ron Musick, iKuni Inc., USA
M. May, GMD, Germany.
S. McClean, University of Ulster, N. Ireland, UK.
E. Neuhold, GMD, Germany.
Y. Pan, Georgia State University, USA
S. Parthasarathy, Ohio State Univ., USA
N. Samatova, Oak Ridge National Labs, USA
D. Skillicorn, Queens Univ., Canada
P. Scheuermann, Northwestern University, USA
K. Sivakumar, Washington State University, USA
N. Soparkar, University of Michigan, USA
D. Talia, DEIS, University of Calabria, Italy
G. Williams, CSIRO, Aust. Nat. Univ., Australia
R. Wirth, Daimler Chrysler, Germany
M. Zaki, RPI, USA
A. Zomaya, University of Western Australia, Australia

Program Highlights

The highlights of this years program include:

- **Keynote:** Delivered by Rajeev Rastogi (Bell Laboratories).
- **Panel Session:** On the topic ''Resource and Location Aware Data Mining''.
- **Contributed Sessions:** On i) high performance data mining systems and and ii) high performance data mining algorithms.

Acknowledgements

Each paper submitted to the workshop was reviewed by at least four members of the international program committee. The organizers would like to thank the international program committee for providing detailed and timely reviews which helped improve the quality of the program. The workshop organizers would also like to thank the SIAM team for helping out with the logistics of producing these proceedings as well as the online support provided.

Detailed Programme

10:15-10:30 **Welcome Address:** Srinivasan Parthasarathy

10:30-12:00 **High Performance Data Mining Algorithms**
Chair: M. Zaki

Principal Component Analysis for Dimension Reduction in Massive Distributed Data Sets
Yongming Qu, George Ostrouchov, Nagiza Samatova, Al Geist
Iowa State University and Oak Ridge National Labs.

A Scalable Multi-Strategy Algorithm for Counting Frequent Itemsets
S. Orlando, P. Palmerini, R. Perego, F. Silvestri
University C' Foscari, CNR-Pisa, University of Pisa

Algorithms for Updating Sequential Patterns
Qingguo Zheng, Ke Xu, Shilong Ma, Weifeng Lu
Beijing University of Aeronautics and Astronautics.

12:00-1:45 Lunch Break

1:45-2:45 **Invited Talk**
Chair: Srinivasan Parthasarathy
Single-Pass Algorithms for Querying and Mining Data Streams
Rajeev Rastogi
Bell Laboratories

2:45 - 3:00 Break

3:00 - 4:00 **High Performance Data Mining Systems**
Chair: Srinivasan Parthasarathy
Using General Grid Tools and Compiler Technology for Distributed Data Mining: Preliminary Report
Wei Du and Gagan Agrawal
Ohio State University

The Design of a Platform for Distributed KDD Components
Patric Wendel and Yi-ke Guo
Imperial College, UK.

4:00-5:30 **Panel Session**
Moderator: Hillol Kargupta
Topic: Resource and Location Aware Data Mining: Challenges and Future Directions
Panelists: TBA

Parthasarathy Srinivasan
2002-03-28

A Scalable Multi-Strategy Algorithm for Counting Frequent Sets

S. Orlando[1], P. Palmerini[1,2], R. Perego[2], F. Silvestri[2,3]

[1]Dipartimento di Informatica, Università Ca' Foscari, Venezia, Italy
[2]Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy
[3]Dipartimento di Informatica, Università di Pisa, Italy

Abstract

In this paper we present DCI (Direct Count & Intersect), a new data mining algorithm for frequent set counting. We also discuss the parallelization strategies used in the design of ParDCI, a distributed and multi-threaded version of DCI. DCI adopts a classical level-wise approach based on candidate generation to extract frequent sets, but uses a hybrid method to determine the supports of candidate itemsets. According to this method, an effective *counting*-based method is exploited during the first iterations, and a fully optimized *intersection*-based technique for the remaining ones. Multiple heuristics strategies are employed by DCI, which is able to adapt its behavior not only to the features of the specific computing platform (e.g. available memory), but also to the features of the dataset being processed. Our approach turned out to be highly scalable and very efficient for mining both short and long patterns from sparse and dense datasets. The experimental results showed that DCI sensibly outperforms FP-growth, a well-known fast algorithm that extracts frequent patterns without candidate generation, and the classical *Apriori* algorithm. We obtained good results for both synthetic and real-world datasets. The large amount of tests conducted permit us to state that the design of DCI is not much focused on specific datasets, and that our optimizations are not over-fitted only to the features of these datasets. ParDCI, the parallel version of DCI, is explicitly devised for targeting clusters of SMP nodes, so that shared memory and message passing paradigms are used at the intra- and inter-node levels, respectively in order to exploit effective parallelization strategies previously proposed for *Apriori*. As a result, ParDCI reaches near optimal speedups.

1 Introduction

Association Rule Mining (ARM), one of the most popular topic in the KDD field [3, 10, 11, 17], regards the extractions of association rules from a database of transactions $\mathcal{D}$. Each rule has the form $X \Rightarrow Y$, where X and Y are sets of items (*itemsets*), such that $(X \cap Y) = \varnothing$. A rule $X \Rightarrow Y$ holds in $\mathcal{D}$ with a minimum confidence c and a minimum support s, if at least the c% of all the transactions containing X also contains Y, and $X \cup Y$ is present in at least the s% of all the transactions of the database. In this paper we are interested in the most computationally expensive phase of ARM, i.e the Frequent Set Counting (*FSC*) one. During this phase, the set of all the *frequent* itemsets is built. An itemset of k items (k-itemset) is frequent if its support is greater than a fixed threshold s, i.e. the itemset occurs in at least *minsup* transactions ($minsup = s/100 \cdot n$, where n is the number of transaction in $\mathcal{D}$).

The computational complexity of the FSC problem derives from the exponential size of its search space $\mathcal{P}(M)$, i.e. the power set of M, where M is the set of items contained in the various transactions of $\mathcal{D}$. A way to prune $\mathcal{P}(M)$ is to restrict the search to itemsets whose subsets are all frequent. The *Apriori* algorithm [6] exactly exploits this pruning technique, thus visiting breadth-first $\mathcal{P}(M)$ for counting itemset supports. At each iteration k, *Apriori* generates C_k, the set of candidate k-itemsets, and counts the occurrences of these candidates in the dataset transactions. The candidates in C_k for which the the minimum support constraint holds are then inserted into F_k, i.e. the set of frequent k-itemsets, and the next iteration is started. Other algorithms [8, 2] adopt instead a depth-first visit of $\mathcal{P}(M)$. In this case the goal is to discover long frequent itemsets first, thus saving the work needed for discovering frequent itemsets included in long

ones. Unfortunately, while it is simple to derive all the frequent itemsets from the maximal ones, the same does not hold for their supports, which require a further counting step. Remember that the exact knowledge of the supports of the frequent itemsets is needed to derive association rule confidences and other measures of interest.

Several variations to the original *Apriori* algorithm, as well as many parallel implementations, have been proposed in the last years. We can recognize two main methods for determining the supports of the various itemsets present in $\mathcal{P}(M)$: a *counting*-based approach [4, 6, 12, 16, 8, 1], and an *intersection*-based one [18, 9, 20]. The former one, also adopted in *Apriori*, exploits a *horizontal* dataset and *counts* how many times each candidate k-itemset occurs in every transaction. The latter method, on the other hand, exploits a *vertical* dataset, where a *tidlist*, i.e. a list of transaction ids, is associated with each item, and itemset supports are determined through tidlist intersections. The *counting*-based approach is, in most cases, quite efficient from the point of view of memory occupation, since only requires to maintain C_k, into the main memory, along with data structures used to quickly access candidates (e.g. hash-trees or prefix-trees). On the other hand, the *intersection*-based method may be much more computational effective than its *counting*-based counterpart [18]. Unfortunately, efficient implementations of *intersection*-based algorithms requires huge amounts of memory to buffer tidlists associated with previously computed frequent $(k-1)$-itemsets.

FP-growth, a completely different algorithm to solve the FSC problem, has recently been proposed by J. Han et al. It is not based on candidate generation, and is currently considered one of the fastest FSC algorithm. It builds in memory a compact representation of the dataset, where repeated patterns are represented once along with the associated repetition counters. The data structure used to store the dataset is called *frequent pattern tree*, or FP-tree for short. The algorithm recursively identifying tree paths which share a common prefix. These paths are intersected by considering the associated counters.

In this paper we discuss in depth DCI (Direct Count & Intersect), a new algorithm to solve the FSC problem. We also discuss a parallel version of DCI, called ParDCI, which is explicitly targeted for clusters of SMPs. As *Apriori*, DCI builds at each iteration the set F_k of the frequent k-itemsets on the basis of C_k. However, DCI adopts a hybrid approach to determine the support of the candidates. In particular, during its first iterations, DCI exploits a novel *counting-based* technique, accompanied by a carefully pruning of the dataset, stored to disk in horizontal form. During the following iterations, DCI adopts a very efficient *intersection-based* technique. DCI starts using this technique as soon as the pruned dataset, whose layout has to be transformed from horizontal into vertical, fits into the main memory of the specific host machine. Tidlists are represented as bit-vectors.

DCI is able to adapt its behavior not only to the features of the specific computing platform, but also to the features of the datasets processed. This ability of DCI is very important, since in the past many novel algorithms were devised, but often they outperformed others only for specific datasets. DCI deals with dataset peculiarities by dynamically choosing among distinct *heuristic strategies*. For example, when a dataset is dense, identical sections appearing in several tidlists are aggregated and clustered, in order to reduce the number of intersections actually performed. Conversely, when a dataset is sparse, the runs of zero bits in the intersected tidlists are promptly identified and skipped.

We will show how the sequential implementation of DCI significantly outperforms previously proposed algorithms. In particular, under a number of different tests and independently of the dataset peculiarities, DCI results to be faster than FP-growth [14]. Moreover, DCI performs very well on both synthetic and real-world datasets characterized by different density features, i.e. datasets from which, due to the different correlations among items, either short or long frequent patterns can be mined.

ParDCI, the parallel version of DCI, adopts different parallelization strategies during the two phases of DCI, i.e. the counting-based and the intersection-based ones. Moreover, these strategies are slightly differentiated with respect to the two levels of parallelism exploited: *intra-node* level within each SMP to exploit shared-memory cooperation, and *inter-node* level among distinct SMPs, where message-passing cooperation is used. Basically, at the inter-node level (coarse grain, message-passing) ParDCI uses a *Count Distribution* technique during the counting-based phase, and a *Candidate Distribution* one during the intersection-based one [5, 13, 19]. The former technique requires the partitioning of the dataset, and the replication of candidates and associated counters. The final values of the counters are derived by all-reducing the various local counters. The latter technique is instead used during the intersection-based phase. It requires an intelligent partitioning of C_k based on the prefixes of itemsets, but a partial/complete replication of the dataset.

This paper is organized as follow. Section 2 discusses the DCI algorithm, while Section 3 sketches the solutions adopted to design ParDCI. In Section 4 we report our experimental results. Finally in Section 5 we present some conclusions and future works.

2 The DCI algorithm

During its initial *counting*-based phase, DCI exploits a *horizontal* layout database with variable length records. DCI, by exploiting effective pruning techniques inspired by DHP [16], trims the transaction database as execution progresses. In particular, a pruned dataset $\mathcal{D}_{k+1}$ is written to the disk at each iteration k, and employed at the next iteration. Let m_k and n_k be the number of items and transactions that are included in the pruned dataset $\mathcal{D}_k$, where $m_k \geq m_{k+1}$ and $n_k \geq n_{k+1}$. Pruning the dataset may thus entail a reduction in I/O activity as the algorithm progresses, but the main benefits come from the reduced computation required for subset counting at each iteration k, due to the reduced number and size of transactions. As soon as the pruned dataset becomes small enough to fit into the main memory, DCI adaptively changes its behavior, builds a *vertical* layout database in-core, and starts adopting an *intersection*-based approach to determine frequent sets. Note, however, that DCI continues to have a level-wise behavior.

At each iteration, DCI generates the candidate set C_k by finding all the pairs of $(k-1)$-itemsets included in F_{k-1} that share a common $(k-2)$-prefix. Since F_{k-1} is lexicographically ordered, the various pairs occur in close positions, and candidate generation is performed with high spatial and temporal locality. Only during the DCI counting-phase, C_k is further pruned by checking whether also all the other subsets of a candidate are included in F_{k-1}. Conversely, during the intersection-based phase, since our intersection method is able to quickly determine the support of a candidate itemsets, we found more profitable to avoid this check.

While during its counting-based phase DCI has to maintain C_k in main memory to search candidates and increment associated counters, this is no longer needed during the intersection-based phase. As soon a candidate k-itemset is generated, DCI determines on-the-fly its support by intersecting the corresponding tidlists. This is an important improvement over other *Apriori*-like algorithms, which suffer from the possible huge memory requirements due to the explosion of the C_k size [14].

DCI makes use of a large body of out-of-core techniques, so that it is able to adapt its behavior also to machines with limited main memory. Datasets are read/written in blocks, to take advantage of I/O prefetching and system pipelining [7]. The outputs of the algorithms, e.g. the various frequent sets F_k, are written to files. These same files are then `mmap`-ped into memory in order to access them during the next iteration for candidate generation.

2.1 Counting-based phase

The techniques used in the counting-based phase of DCI are detailed in [15], where the same authors proposed an effective algorithm for miming short patterns. Since the counting-based approach is used only for few iterations (in all the experiments conducted DCI starts using intersections at the third or fourth iteration), in the following we only sketch the main features of the counting method adopted.

In the first iteration, as all FSC algorithms, DCI exploits a vector of counters, which are directly addressed through item identifiers. For $k \geq 2$, instead of using complex data structures like hash-trees or prefix-trees, DCI uses a novel *Direct Count technique* that can be thought as a generalization of the technique used for $k = 1$. The technique uses a *prefix table*, $\text{PREFIX}_k[\,]$, of size $\binom{m_k}{2}$. In particular, each entry of $\text{PREFIX}_k[\,]$ is associated with a distinct *ordered prefix* of two items. For $k = 2$, $\text{PREFIX}_k[\,]$ can directly contain the counters associated with the various candidate 2-itemsets, while, for $k > 2$, each entry of $\text{PREFIX}_k[\,]$ contains the pointer to the contiguous section of ordered candidates in C_k sharing the same prefix. To permit the various entries of $\text{PREFIX}_k[\,]$ to be directly accessed, we devised an order preserving, minimal perfect hash function. This prefix table is thus used to count the support of candidates in C_k as follows. For each transaction $t = \{t_1, \ldots, t_{|t|}\}$, we select all the possible 2-prefixes of all k-subsets included in t. We then exploit $\text{PREFIX}_k[\,]$ to find the sections of C_k which must be visited in order to check set-inclusion of candidates in transaction t.

2.2 Intersection-based phase

Since the counting-based approach becomes less efficient as k increases [18], DCI starts its intersection-based phase as soon as possible. Unfortunately, the intersection-based method needs to maintain in memory the vertical representation of the pruned dataset. So, at iteration k, $k \geq 2$, DCI checks whether the pruned dataset $\mathcal{D}_k$ may fit into the main memory. When the dataset becomes small enough, its vertical in-core representation is built on the fly, while the transactions are read and counted against C_k. The *intersection-based* method thus starts at the next iteration.

The vertical layout of the dataset is based on fixed length records (tidlists), stored as *bit-vectors*. The whole vertical dataset can thus be seen as a bidimensional bit-array $\mathcal{VD}[\,][\,]$, whose rows correspond to the bit-vectors associated with not pruned items. Therefore, the amount of memory required to store $\mathcal{VD}[\,][\,]$ is $m_k \times n_k$ bits.

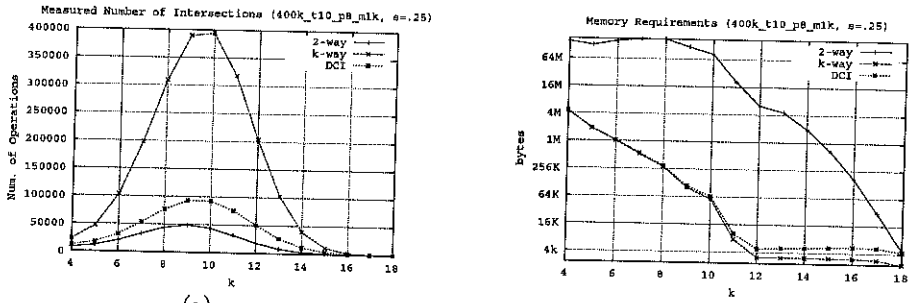
At each iteration of its intersection-based phase, DCI computes F_k as follows. For each candidate itemset $c \in C_k$, we *and*-intersect the k bit-vectors associated with the items included in c (k-way intersection), and count the 1's present in the resulting bit-vector . If this number is $\geq$ *minsup*, we insert c into F_k. Consider that a bit-vector intersection can be carried out very efficiently and with high spatial locality by using primitive Boolean *and* instructions with word operands. As previously stated, this method does not require C_k to be kept in memory: we can compute the support of each candidate c on-the-fly, as soon as it is generated.

The strategy above is, in principle, highly inefficient, because it always needs a k-way intersection to determine the support of each candidate c. Conversely, if we had enough memory to maintain the tidlists (bit-vectors) associated with all the frequent $(k-1)$-itemsets in F_{k-1}, we could carry out the same computation through a single 2-way intersection. Unfortunately, a pure 2-way intersection approach does not scale, due to the huge amount of memory required. Nevertheless, a caching policy could be exploited in order to save work and speed up our k-way intersection method. To this end, DCI uses a small "cache" buffer to store the results of all the $k-2$ intermediate intersections that have been computed to determine the support of the last candidate that has been evaluated. Since candidate itemsets are generated in lexicographic order, with high probability two consecutive candidates, e.g. c and c', share a common prefix. Suppose that c and c' share a prefix of length $h \geq 2$. When we process c', we can avoid to perform the first $h-1$ intersections since their result can be found in the cache.

To evaluate the effectiveness of our caching policy, we counted the actual number of intersections carried out by DCI on a synthetic dataset. We compared this number with the best and the worst case. The former corresponds to the adoption of a 2-way intersection approach, which is only possible if we can fully cache the tidlists associated with all the frequent $(k-1)$-itemsets in F_{k-1}. The latter case regards the adoption of a pure k-way intersection method, i.e. a method that does not exploit caching at all. Figure 1.(a) plots the results of this analysis for support threshold $s = 0.25\%$. The caching policy of DCI turns out to be very effective, since the actual number of intersections performed results to be very close to the best case. Moreover, memory requirements for the three approaches are plotted in Figure 1.(b). As expected, DCI requires orders of magnitude less memory than a pure 2-way intersection approach, thus better exploiting memory hierarchies.

Others heuristics are used within DCI to further reduce intersection costs. More specifically, two different optimization techniques are exploited for *sparse* and *dense* datasets. In order to apply the right optimization, the vertical dataset is tested for checking its density as soon as it is built. To this end we compare the bit vectors associated with the *most frequent items*, i.e., the vectors which need to be intersected several times since the associated items occur in many candidates. If large sections of these bit-vectors turns out to be identical, we deduce that the dataset is dense and we adopt a specific heuristics which exploits similarities between these vectors. Otherwise the technique for sparse datasets is adopted.

- **Sparse datasets.** Sparse or moderately dense datasets originate bit-vectors containing long runs of 0's. To speedup computation, while we compute the intersection of the bit vectors relative to the first two items c_1 and c_2 of a generic candidate itemset $c = \{c_1, c_2, \ldots, c_k\} \in C_k$, we also identify and maintain information about the the runs of 0's appearing in the resulting bit vector stored in cache. Then, further intersections that are needed to determine the support of c (as well as intersections needed to process other k-itemsets sharing the same 2-item prefix) will skip these runs of 0's, so that



(a) (b)

Figure 1: Per iteration number of tidlist intersections performed (a), and memory requirements (b), for DCI, and the pure 2-way and k-way intersection-based approaches.

only vector segments which may contain 1's are actually intersected. Since information about the runs of 0's are computed once, and the same information is reused many times, this optimization results to be very effective.

Moreover, sparse and moderately dense datasets offer the possibility of further pruning vertical datasets as computation progresses. The benefits of pruning regard the reduction in the length of the bit vectors and thus in the cost of intersections. Note that a transaction, i.e. a column of $\mathcal{VD}$, can be removed from the vertical dataset when it does not contain any of the itemsets included in F_k. This check can simply be done by *or*-ing the intersection bit-vectors computed for all the frequent k-itemsets. However, we observed that dataset pruning is expensive, since vectors must be compacted at the level of single bits. Hence DCI prunes the dataset only if turns out to be profitable, i.e. if we can obtain a large reduction in the vector length, and the number of vectors to be compacted is small with respect to the cardinality of C_k.

- **Dense datasets.** If the dataset turns out to be dense, we expect to deal with a dataset characterized by strong correlations among the most frequent items. This not only means that the bit-vectors associated with the *most frequent items* contain long runs of 1's, but also that they turn out to be very similar. The heuristic technique adopted by DCI for dense dataset thus works as follows:
  - reorder the columns of the vertical dataset, in order to move identical segments of the bit vectors associated with the most frequent items to the first consecutive positions;
  - since each candidate likely includes several of these most frequent items, avoid repeatedly intersecting the identical segments of the corresponding vectors. This technique may save a lot of work because (1) the intersection of identical vector segments is done once, (2) the identical segments are usually very large, and (3) long candidate itemsets likely contains several of these most frequent.

3 ParDCI

In the following we describe the different parallelization techniques exploited for the *counting*- and *intersection*-based phases of ParDCI, the parallel version of DCI. Since our target architecture is a cluster of SMP nodes, in both phases we distinguish between *intra-node* and *inter-node* levels of parallelism. At the inter-node level we used the message–passing paradigm through the MPI communication library, while at the intra-node level we exploited multi-threading through the *Posix Thread* library. A *Count Distribution* approach is adopted to parallelize the *counting*-based phase, while the *intersection*-based phase exploits a very effective *Candidate Distribution* approach [5].

3.1 The counting-based phase

At the inter-node level, the dataset is statically split in a number of partitions equal to the number of SMP nodes available. The sizes of partitions depend on the relative powers of nodes. At each iteration k, a identical copy of C_k is generated independently by each node. Then each node p reads blocks of transactions from its own dataset partition $\mathcal{D}_{p,k}$, performs subset counting, and writes pruned transactions to $\mathcal{D}_{p,k+1}$. At the end of the iteration, an all-reduce operation is performed to update the counters associated to all candidates of C_k, and all the nodes produce an identical set F_k.

At the intra-node level each node uses a pool of threads. They have the task of checking in parallel each of the candidate itemset against chunks of transactions read from $\mathcal{D}_{p,k}$. The task of subdividing the local dataset into disjoint chunks is assigned to a particular thread, the *Master Thread*. It loops reading blocks of transactions and forwarding them to the *Worker Threads* executing the counting task. To overlap computation with I/O, minimize synchronization, and avoid unnecessary data copying overheads, we used an optimized producer/consumer schema for the cooperation among the Master and Worker threads. Through two shared queues, the Master and Worker threads cooperate by exchanging pointers to empty and full buffers storing block of transactions to be processed.

When all transactions in $\mathcal{D}_{p,k}$ have been processed by a node p, the corresponding Master thread performs a local reduction operation over the thread-private counters (reduction at the intra-node level), before performing via MPI the global counter reduction operation with all the other Master threads running on the other nodes (reduction at the inter-node level). Finally, to complete the iteration of the algorithm, each Master thread generates and writes F_k to the local disk.

3.2 The intersection-based phase

During the intersection-based phase, a Candidate Distribution approach is adopted at both the inter- and intra-node levels. This parallelization schema makes the parallel nodes completely independent: inter-node communications are no longer needed for all the following iterations of ParDCI. Let us first consider the inter-node level, and suppose that the intersection-based phase is started at iteration $\overline{k}+1$. Therefore, at iteration $\overline{k}$ the various nodes build on-the-fly the bit-vectors representing their own in-core portions of the vertical dataset. Before starting the intersection-base phase, the partial vertical datasets are broadcast to obtain a complete replication of the whole vertical dataset on each node.

The frequent set $F_{\overline{k}}$ (i.e., the set computed in the last counting-based iteration) is then statically partitioned by exploiting problem-domain knowledge. A disjoint partition $F_{p,\overline{k}}$ of $F_{\overline{k}}$ is assigned to each node p, where $\bigcup_p F_{p,\overline{k}} = F_{\overline{k}}$. It is worth remarking that this partitioning entails a Candidate Distribution schema for all the following iterations, according to which each node p will be able to generate a unique C_k^p $(k > \overline{k})$ independently of all the other nodes, where $C_k^p \cap C_k^{p'} = \emptyset$ if $p \neq p'$, and $\bigcup_p C_k^p = C_k$.

$F_{\overline{k}}$ is partitioned as follows. First, it is split into l *sections* on the basis of the prefixes of the lexicographically ordered frequent itemsets included. All the frequent $\overline{k}$-itemsets that share the same $\overline{k}-1$ prefix are assigned to the same section. Since ParDCI builds each candidate $(\overline{k}+1)$-itemsets as the union of two frequent $\overline{k}$-itemsets sharing the first $\overline{k}-1$ items, we are sure that each candidate can independently be generated starting from one of the l disjoint sections of $F_{\overline{k}}$. Then the various partitions $F_{p,\overline{k}}$ are created by assigning round-robin the l sections to the np processing nodes. Since $l >> np$, this round-robin policy well balances the workload at the inter-node level. Once completed the partitioning of $F_{\overline{k}}$, the nodes independently generate the associated candidates and determine their supports by intersecting the corresponding tidlists of the replicated vertical dataset. Nodes continue to work according to the schema above also for the following iterations, without any communication exchange.

At the intra-node level, a similar Candidate Distribution approach is employed, but at a finer granularity by using dynamic scheduling to ensure load balancing. In particular, at each iteration k the Master thread of a node p initially splits the local partition of $F_{p,k-1}$ into x disjoint partitions S_i, $i = 1, \ldots, x$, where $x >> t$, and t is the number of active threads. The boundaries of these x partitions are then inserted in a shared queue. Once the shared queue is initialized, also the Master thread becomes a Worker. Thereinafter, each Worker thread loops and self-schedules its work by performing the following steps:

Table 1: Datasets used in the experiments.

Dataset	Description
T25I10D10K	1K items and 10K transactions. The average size of transactions is 25, and the average size of the maximal potentially frequent itemsets is 10. Synthetic dataset available at `http://www.cs.sfu.ca/~peijian/personal/publications/T25I10D10k.dat.gz`
T25I20D100K	10K items and 100K transactions. The average size of transactions is 25, and the average size of the maximal potentially frequent itemsets is 20. Synthetic dataset available at `http://www.cs.sfu.ca/~peijian/personal/publications/T25I20D100k.dat.gz`
400k_t10_p8_m10k	10K items and 400K transactions. The average size of transactions is 10, and the average size of the maximal potentially frequent itemsets is 8. Synthetic dataset created with the IBM dataset generator [6].
400k_t30_p16_m1k	1K items and 400K transactions. The average size of transactions is 30, and the average size of the maximal potentially frequent itemsets is 16. Synthetic dataset created with the IBM dataset generator [6].
t20_p8_m1k	With this notation we identify a series of synthetic datasets characterized by 1K items. The average transaction size is 20, and the average size of maximal potentially frequent itemsets is 8. The number of transactions is varied for scaling measurements.
t50_p32_m1k	A series of three synthetic datasets with the same number of items (1K), average transaction size of 50, and average size of maximal potentially frequent itemsets equal to 32. We used three datasets of this series with 1000k, 2000k and 3000k transactions.
connect-4	Dense dataset with 130 items and about 60K transactions. The maximal transaction size is 45. Available at `http://www.cs.sfu.ca/~wangk/ucidata/dataset/connect-4/connect-4.data`
BMS-WebView-1	497 items and 59K transactions containing click-stream data from an e-commerce web site. Each transaction is a web session consisting of all the product detail pages viewed in that session. Available at `http://www.ecn.purdue.edu/KDDCUP/data/BMS-WebView-1.dat.gz`

1. access in mutual exclusion the queue and extract information to get S_i, i.e. a partition of the local $F_{p,k-1}$. If the queue is empty, write $F_{p,k}$ to disk and start a new iteration.
2. generate a new candidate k-itemset c from S_i. If it is not possible to generate further candidates, go to step 1.
3. compute on-the-fly the support of c by intersecting the vectors associated to the k items of c. In order to reuse effectively previous work, each thread exploits a private cache for storing the partial results of intersections (see Section 2.2). If c turns out to be frequent, put c into $F_{p,k}$. Go to step 2.

4 Experimental Results

The DCI algorithm is currently available in two versions, a MS-Windows one, and a Linux one. ParDCI, which exploits the MPICH MPI and the *pthread* libraries, is currently available only for the Linux platform.

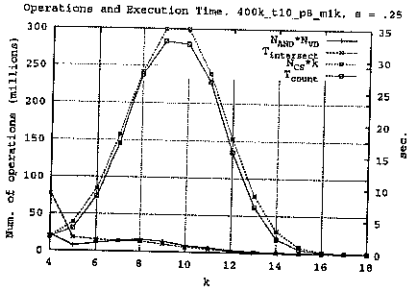


Figure 2: Theoretical and measured computational costs of intersection-based and counting-based iterations on dataset 400k_t10_p8_m1k ($s = 0.25\%$). Times $T_{intersect}$ and T_{count} are measured in seconds according to the scale on the right hand y axis. Millions of operations are instead reported on the left hand y axis.

We used the MS-Windows version of DCI to compare its performance with other FSC algorithms. For test comparisons we used the FP-growth algorithm, currently considered one of the fastest algorithm for FSC[1], and the Christian Borgelt's implementation of *Apriori*[2]. For the sequential tests we used a Windows-NT workstation equipped with a Pentium II 350 MHz processor, 256 MB of RAM memory and a SCSI-2 disk. For testing ParDCI performance, we employed a small cluster of three Pentium II 233MHz two-way SMPs, for a total of six processors. Each SMP is equipped with 256 MBytes of main memory and a SCSI disk. For the tests, we used both synthetic and real datasets by varying the minimum support threshold s. The characteristics of the datasets used are reported in Table 1.

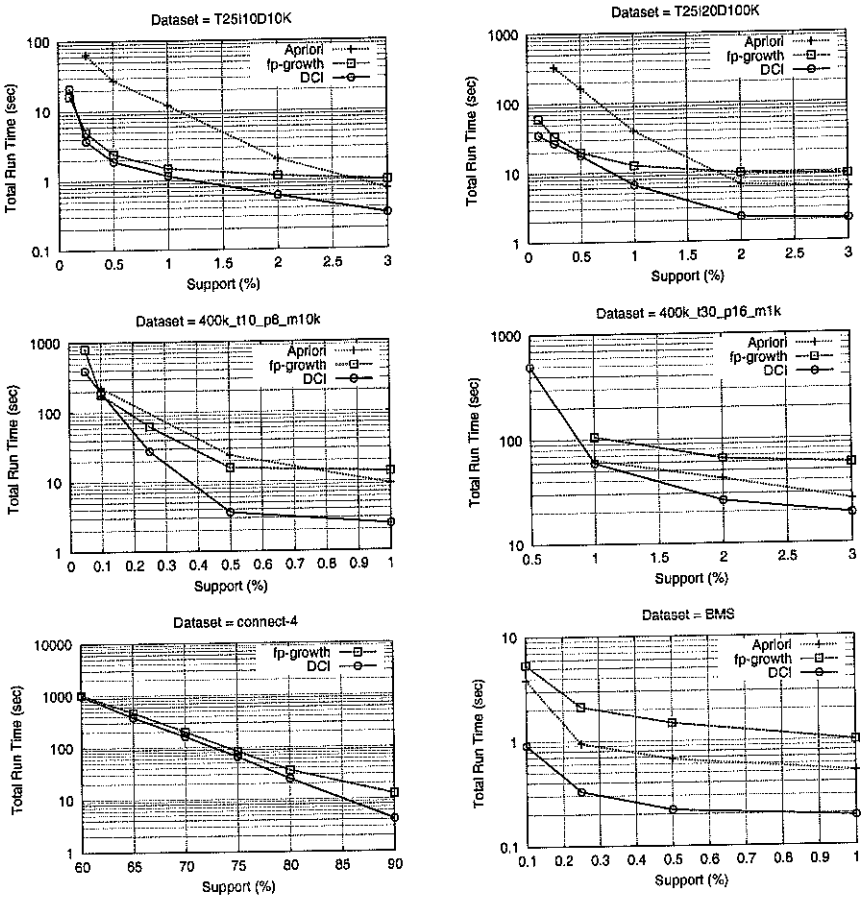


Figure 3: Total execution times for DCI, *Apriori*, and FP-growth on various datasets as a function of the support threshold.

Computational cost of intersection-based phase. We analyzed the advantages of adopting the intersection-based approach, over the exploitation of the counting-based approach for all the iterations of the algorithm. The computational costs of each DCI counting-based iteration is dominated by subset counting. Due to our 2-item prefix table, which allows us to directly select a section of C_k with a common prefix, at most $k-2$ comparisons are necessary in order to check whether a given candidate k-itemset is included within a

[1]We acknowledge Prof. Jiawei Han for kindly providing us the latest binary version of FP-growth. This version of FP-growth was sensible optimized with respect to the one used for the tests reported in [14].

[2]`http://fuzzy.cs.uni-magdeburg.de/~borgelt`

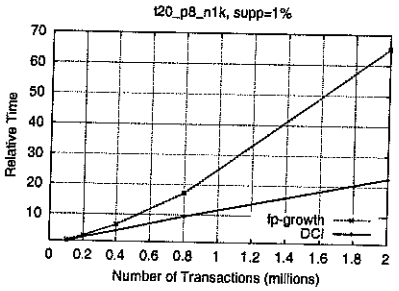


Figure 4: Relative execution times on datasets in the series t20_p8_m1k ($s = 1\%$) when varying the number of transactions (from $100K$ to $2M$).

transaction t. Hence, the number of operations performed at iteration k is approximately $T_{count} = O(N_{CS} \cdot k)$, where N_{CS} is the total number of candidates actually visited for counting the supports of all the transactions in $\mathcal{D}_k$.

On the other hand, the computational cost of each DCI intersection-based iteration is proportional to the number of *and* operations needed to determine the supports of all candidate itemsets. The number of *and* depends on both the average length of tidlists and the number of candidate itemsets. Therefore, the number of operations actually performed by DCI at iteration k is approximatively $T_{intersect} = O(N_{AND} \cdot N_{VD})$, where N_{AND} is the total number of tidlist pairs actually intersected, while N_{VD} is the average number of operations needed for *and*-ing a pair of tidlists. In principle we can say that N_{VD} depends on the average length of tidlists, but we have to consider that DCI exploits several optimizations aimed to reducing the number of operations actually performed (see Section 2.2).

This simple analysis is confirmed by our experimental evaluation. In Figure 2 the measured per-iteration execution times, i.e. T_{count} and $T_{intersect}$, are plotted against their analytic estimates as a function of the iteration index k. The dataset considered was 400k_t10_p8_m1k, mined with a support threshold equal to $s = 0.25\%$. The actual values of N_{CS}, N_{AND} and N_{VD} were determined by profiling execution.

DCI performances and comparisons. Figure 3 reports the total execution times obtained running *Apriori*, FP-growth, and DCI on the datasets described in Table 1 as a function of the support threshold s. In all the tests conducted, DCI outperforms FP-growth with speedups up to 8. Of course, DCI also remarkably outperforms *Apriori*, in some cases for more than one order of magnitude. For connect-4, the dense dataset, the curve of *Apriori* is not shown, due to the relatively too long execution times. Note that, accordingly to [21], on the real-world dataset BMS, *Apriori* turned out to be slightly faster than FP-growth.

The encouraging results obtained with DCI are due to both the efficiency of the counting method exploited during early iterations, and the effectiveness of the intersection-based approach used when the pruned vertical dataset fits into the main memory. For only a dataset, namely T25I10D10K, FP-growth turns out to be slightly faster than DCI for $s = 0.1\%$. The cause of this behavior is the size of C_3, which in this specific case results much larger than the actual size of F_3. Hence, DCI has to carry out a lot of useless work to determine the support of many candidate itemsets, which will eventually result to be not frequent. In this case the FP-growth is faster than DCI since it does not require candidates generation.

We also tested the scale-up behavior of DCI when the size of the dataset is increased. The various dataset samples employed for the tests belong to the series t20_p8_m1k. Figure 4 plots the execution times of FP-growth and DCI as a function of the number of transactions contained in the dataset processed, by keeping constant $s = 1\%$. The times reported are normalized with respect to the execution time of DCI on the smallest dataset sample of $100k$ transactions. DCI scales much better than FP-growth: its execution time is a liner function of dataset size. For example, to process the dataset with 2 millions of transactions, DCI requires about 22 times the execution time spent on $100k$ transactions, while for FP-growth this ratio is about 65.

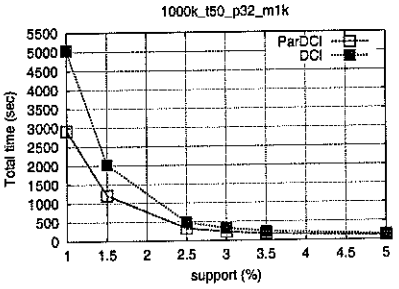


Figure 5: Sequential and multithreaded execution times for dataset 1000K as a function of s.

Performance evaluation of ParDCI. For these tests we used the synthetic dataset series identified as $t50\_p32\_m1k$ in Table 1. We varied the total number of transactions from 1000k to 3000k. In the following we will identify the various synthetic datasets on the basis of their number of transactions, i.e. 1000k, 2000k, and 3000k.

First we measured execution times of ParDCI on dataset 1000k. In this test we only used a single 2-way SMP node, in order to compare DCI with ParDCI, which in this case only exploits multi-threading. Figure 5 plots the total execution times as a function of the support thresholds s. The reduction in the total execution time is quasi optimal (nearly optimal speedup) for support thresholds that involve expensive computations.

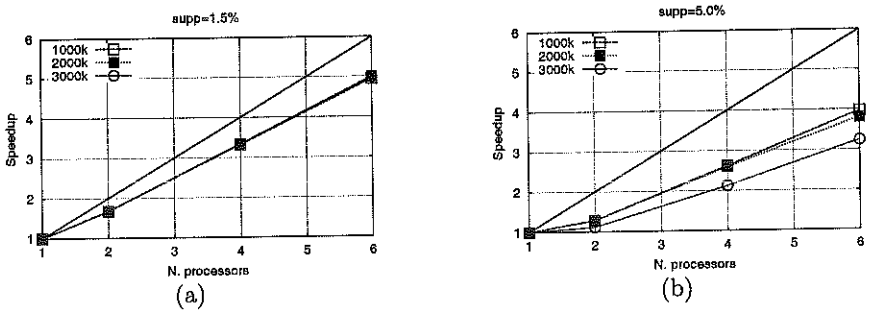


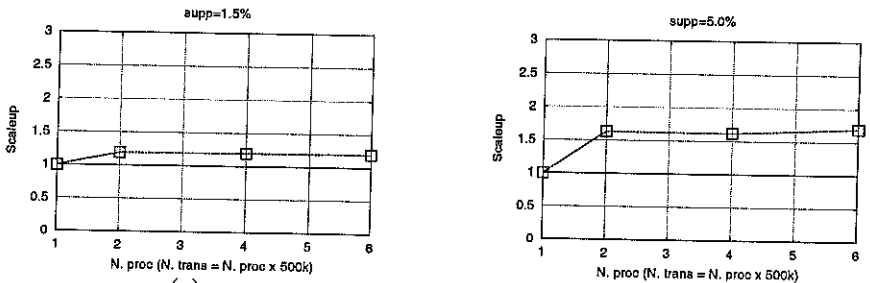
Figure 6: Speedup for datasets 1000K, 2000K and 3000K with $s = 1.5\%$ (a) and $s = 5\%$(b).

Figure 6 plots the speedups obtained on the three synthetic datasets for two fixed support thresholds ($s = 1.5\%$ and $s = 5\%$), as a function of the number of processors used. Consider that, since our cluster is composed of three 2-way SMPs, we mapped tasks on processors always using the minimum number of nodes (e.g., when we used 4 processors, we actually employed 2 SMP nodes). This implies that experiments performed on either 1 or 2 processors actually have identical memory and disk resources available, whereas the execution on 4 processors benefit from a double amount of such resources.

According to our tests, ParDCI showed a speedup close to the optimal one. Considering the results obtained with one or two processors, one can note that the slope of the speedup curve is relatively worse than its theoretical limit, due to resource sharing and thread implementation overheads at the inter-node level. Nevertheless, when additional nodes are employed, the slope of the curve improves. For all the three datasets, when $s = 5\%$, a very small number of frequent itemsets is obtained. As a consequence, the CPU-time decreases, and becomes relatively smaller than I/O and also interprocess communication times.

Finally, Figure 7 plots the scaleup, i.e. the relative execution times measured by varying, at the same time, the number of processors and the dataset size. We can observe that the scaling behavior remains constant, although slightly worse than the theoretical limit.

The strategies adopted for partitioning dataset and candidates on our homogeneous cluster of SMPs sufficed for balancing the workload. In out tests we observed very limited imbalance, below 0.5%. For



(a) (b)

Figure 7: Scaling behavior obtained varying the dataset size along with the processor number for $s = 1.5\%$ (a) and $s = 5\%$(b).

targeting heterogeneous or non-dedicated clusters we plan to introduce in ParDCI adaptive partitioning strategies at the inter-node level.

5 Conclusions and Future Works

DCI and ParDCI use different approaches for extracting frequent patterns: *counting*-based during the first iterations and *intersection*-based for the following iterations. One of the main innovative features of the two algorithms regards the ability to apply different heuristic strategies on the basis of the characteristics of a specific dataset. Different techniques are thus used for dense and sparse datasets during the intersection-based phase. These techniques are able to strongly reduce the complexity of intersections. Unlike other algorithms, such as maximal frequent set ones, even for dense datasets, from which very long patterns can be extracted, we are able to determine the exact support of frequent itemsets. Another important feature of DCI and ParDCI is the ability to adapt their behaviors to the characteristics of the specific computing platform. For example, the intersection-based phase is started only if the vertical layout representation of the dataset can be stored into the main memory. Since our pruning technique strongly reduce dataset size as counting-based iterations progress, in our tests, performed on machines with only 256MB of main memory, the optimized intersection-based phase always started at the third or fourth iteration.

In order to analyze the scalability of our algorithms, we have to consider their memory requirements. When DCI builds the in-core vertical representation of the pruned dataset, say at iteration $\overline{k}$, it needs enough memory to store both the dataset and $C_{\overline{k}}$. Conversely, since ParDCI uses a Count Distribution approach for parallelization, the per-node memory requirement at iterations $\overline{k}$ is lower. In fact each SMP node has only to build a vertical representation of its own partition of the pruned dataset. ParDCI will create a complete in-core vertical dataset at the next iteration $\overline{k}+1$, by joining the various partitions. Moreover, during a generic intersection-based iteration k, both algoriths do not keep candidates in-core. DCI generates candidates by accessing with high locality the mmap-ped file containing F_{k-1}, while their supports are computed on-the-fly. In ParDCI, since a Candidate Distribution technique is adopted, each node need to access only a partition of F_{k-1}.

As a result of its optimized design, DCI significantly outperformed *Apriori* and FP-growth. For many datasets the performance improvements were impressive. The results were very good, independently of the support threshold, not only for synthetic datasets, but also for real-world datasets. The variety of datasets used and the large amount of tests conducted permit us to state that the design of DCI is not focused on specific datasets, and that our optimizations are not over-fitted only to the features of these datasets [21].

ParDCI, the multi-threaded and distributed version of DCI, is able to effectively exploit our cluster of SMPs, thus exhibiting excellent scaleups and speedups. Our implementation of the Count and Candidate Distribution strategies for parallelization, used at both inter and intra-node levels, resulted to be very effective with respect to load balancing and communication overhead minimization. In the near future we plan to extend ParDCI with adaptive *work stealing* policies aimed to efficiently exploit heterogeneous/grid

environments. To share our efforts with the data mining community, we made DCI and ParDCI binary codes available for research purposes at `http://www.miles.cnuce.cnr.it/~palmeri/datam/DCI`.

References

[1] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets. *Journal of Parallel and Distributed Computing.* To appear.

[2] R. C. Agarwal, C. C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In *Proc. of the 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, pages 108–118, 2000.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining Association between Sets of Items in Massive Databases. In *ACM-SIGMOD 1993 Int'l Conf. on Management of Data*, pages 207–216. ACM, 1993. Washington D.C. USA.

[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast Discovery of Association Rules in Large Databases. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.

[5] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Transaction On Knowledge And Data Engineering*, 8:962–969, 1996.

[6] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. of the 20th VLDB Conf.*, pages 487–499, 1994.

[7] R. Baraglia, D. Laforenza, S. Orlando, P. Palmerini, and R. Perego. Implementation issues in the design of I/O intensive data mining applications on clusters of workstations. In *Proc. of the 3rd Work. on High Performance Data Mining, (IPDPS-2000), Cancun, Mexico*, pages 350–357. LNCS 1800 Spinger-Verlag, 2000.

[8] R. J. Bayardo Jr. Efficiently Mining Long Patterns from Databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 85–93, Seattle, Washington, USA, 1998.

[9] Brian Dunkel and Nandit Soparkar. Data organization and access for efficient data mining. In *Proc. of the 15th ICDE Int. Conf. on Data Engineering*, pages 522–529, Sydney, Australia, 1999. IEEE Computer Society.

[10] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smith, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI Press, 1998.

[11] V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining Very Large Databases. *IEEE Computer*, 32(8):38–45, 1999.

[12] E. H. Han, G. Karypis, and Kumar V. Scalable Parallel Data Mining for Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):337–352, May/June 2000.

[13] Eui-Hong (Sam) Han, George Karypis, and Vipin Kumar. Scalable Parallel Data Mining for Association Rules. *IEEE Transaction on Knowledge and Data Engineering*, 12(3):337–352, may/june 2000.

[14] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Dallas, Texas, USA, 2000.

[15] S. Orlando, P. Palmerini, and R. Perego. Enhancing the Apriori Algorithm for Frequent Set Counting. In *Proc. of the 3rd Int. Conf. on Data Warehousing and Knowledge Discovery, DaWaK 2001, LNCS 2114*, pages 71–82, Munich, Germany, 2001.

[16] J. S. Park, M.-S. Chen, and P. S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. In *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 175–186, 1995.

[17] N. Ramakrishnan and A. Y. Grama. Data Mining: From Serendipity to Science. *IEEE Computer*, 32(8):34–37, 1999.

[18] A. Savasere, E. Omiecinski, and S. B. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proc. of the 21th VLDB Conf.*, pages 432–444, Zurich, Switzerland, 1995.

[19] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, 1999.

[20] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, May/June 2000.

[21] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In *Proc. of KDD-2001*, 2001.