POLITECNICO DI TORINO

III Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Master of Science Thesis

Index support for itemset mining

Advisors:
prof. Elena Baralis
ing. Tania Cerquitelli

Candidate:
Alberto Grand

November 2009

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The research activity in data mining has been initially focused on discovering efficient in-memory algorithms to perform the task of frequent itemset mining. However, many of them scale non-linearly with the size of the datasets and, as the latter become larger, the mining process is faced with issues such as main memory exhaustion, massive I/O and long execution times.

As a consequence, approaches that rely on secondary storage to support the data mining process are drawing increasing attention. Many of the proposed solutions still suffer from some limitations: they often address specific data distributions, they can hardly handle more than a few million records and their performance is usually worse than flat file mining.

This master thesis has been pursuing the development of a disk-based data mining-oriented framework initiated by former graduate students. The proposed approach splits the mining process in two steps. A complete, compact and persistent representation of the original dataset is built once, stored on secondary memory in the form of a hybrid prefix-tree (called Hybrid-Tree) and exploited for multiple mining sessions. Mining activities are then performed on relevant portions of the data structure. In addition, the framework benefits from an ad-hoc buffer management strategy which efficiently handles data loading from secondary to main memory.

The focus of my work has been on different optimizations which proved able to significantly improve both the creation and the mining process in terms of execution time, disk space usage,

main memory consumption and scalability. The proposed contributions have been addressing the materialization process, the development of an indexing structure and the integration of an efficient mining algorithm in the disk-based framework.

Building the persistent hybrid tree requires three main steps: (i) sort the transactional dataset, (ii) build a temporary disk-based tree structure and (iii) store the final Hybrid-Tree on disk. Sorting transactions beforehand yields important benefits in the creation phase, because it avoids multiple, temporally non-local visits of the same portion of the tree, thus reducing I/O times. However, since this step was the most resource-demanding one (i.e., 55%-60% of the overall creation time), a novel sorting technique has been devised. The proposed solution relies on splitting the dataset into disjoint, fixed-size partitions, which are then sorted and written on disk independently of all others. This is a significant benefit, because it avoids several passes over the entire dataset and, seen the huge size of the latter, frequent memory swaps. A further achievement of the proposed approach is that it scales linearly with the size of the dataset.

To enhance mining activity, an indexing structure has been designed and developed for the Hybrid-Tree. This compact index enables an item-based traversal of the tree, required by several frequent itemset mining algorithms. The index consists of a per-item array-based structure, allowing fast loading and scanning. Although non-covering with respect to the mining task, this index replicates a considerable amount of the information included in the tree, which often results in a larger size than the tree itself. Hence, a compression technique has been designed and implemented to reduce its size, exploiting a differential coding scheme and ad-hoc

structures that maximize data packing. This approach was able to yield a disk space saving of about 33%, while additionally being roughly linear in the size of the index.

Further, some primitives have been developed to support the selective retrieval from the Hybrid-Tree of the transactional data involved in the mining process. They rely on the Item-Index to carry out the extraction of item-based projections of the dataset by efficiently navigating the tree in both top-down and bottom-up fashion. Their main advantage is that of enabling a tighter integration with mining algorithms. In particular, LCM v.2 is a mining algorithm which has proved to outperform all other state-of-the-art algorithms in most cases. Despite its linear complexity, which makes it very suitable for large datasets, its original implementation requires that the entire dataset be loaded in main memory, thus majorly limiting its scalability.

Following a tight integration approach, LCM v.2 has been analyzed to identify algorithm-specific data access behaviors. Its routines have been decomposed and interleaved with the selective retrieval of item-based projections from the Hybrid-Tree. This was possible thanks to the developed data access primitives.

The algorithm has thus been integrated in our framework by partitioning the loading so that only a support-based projection of a single item is processed in main memory at a time. Its memory requirements have thus been greatly reduced, ultimately making it far more scalable. Performance assessments showed that the mining process still completes in a relatively short time (less than 45 minutes) on datasets larger than the available physical memory by over one order of magnitude, while memory usage is still very low.

CHAPTER 1

INTRODUCTION

The field of knowledge discovery and data mining is targeted at the extraction of useful information from raw data. The researcher community has shown increasing interest in this challenge, spurred by the explosion, in the recent years, of the amount of information stored in large databases, such as data warehouses.

One of the activities of data mining is known as association rule mining, which heavily relies on frequent itemset mining. This is a powerful method aimed at finding correlations among data items in a transactional dataset, given a support threshold. It is widely used in the so-called market basket analysis to discover regularities in the shopping behavior of customers of supermarkets, on-line shops and the like, but it can be profitably applied to a number of other tasks such as credit card fraud detection, medical diagnosis and monitoring, network traffic characterization.

The research activity in data mining has been initially focused on the discovery of efficient algorithms to perform the resource-intensive task of frequent itemset mining. Nonetheless, many of these algorithms are computationally complex and scale non-linearly with the size of the datasets. As datasets become larger, the number of frequent patterns explodes and the mining process is faced with critical issues such as main memory exhaustion, massive I/O and very long execution times.

While much research in this area is still largely focused on the development of main-memory data structures that enable a compact representation and an efficient analysis of transactional data, approaches that rely on secondary storage to support the effective retrieval of data involved in the mining process are receiving increasing attention.

Up to now, many of the proposed solutions still suffer from some limitations: they often address specific data distributions, they can hardly handle more than a few millions of records and their performance is usually worse than, or at best comparable with flat file mining.

The aim of this master thesis has been that of pursuing the development of a disk-based data mining-oriented framework initiated by former graduate students.

This framework exploits a disk-resident, permanent data structure which compactly and thoroughly represents the original dataset in the form of a prefix-tree, called Hybrid-Tree, similar to the FP-tree structure employed by the FP-growth mining algorithm. The proposed approach splits the mining process in two steps. First, a persistent representation of the original dataset is built once, stored on secondary memory and exploited for multiple mining sessions. Mining activities are then performed on relevant portions of the data structure.

In addition, this framework benefits from an ad-hoc buffer management strategy which efficiently handles the loading and unloading of selected pages from the tree in and out of main memory.

The focus of my work has been on different optimizations which proved able to significantly improve both the creation and the mining process, in terms of execution time, disk space usage, main memory consumption and scalability.

My contribution has been addressing the following issues:

- creation of the persistent representation;
- development of an indexing structure;
- integration of an efficient mining algorithm in the disk-based framework.

The rest of this master thesis work is organized as follows.

Chapter 2 gives an overview of some efficient state-of-the-art mining algorithms; it then introduces the problem of large dataset mining and presents a number of possible approaches to deal with it.

Chapter 3 describes the proposed disk-based Hybrid-Tree approach to support itemset extraction from very large datasets.

The steps necessary to the creation of the Hybrid-Tree and the issues that may arise are explained in Chapter 4, while the low-level data access framework exploited in both the creation and the mining phase is described in Chapter 5.

Chapter 6 presents two possible strategies to integrate existing mining algorithms into the developed framework. It then focuses on one efficient mining algorithm and describes how it was tightly integrated in the framework to enhance its scalability.

Chapter 7 presents an analysis of the compaction achieved by the Hybrid-Tree and Item-Index structures. Experimental results, in terms of materialization time, are then presented to evaluate the benefits provided by the devised sorting algorithm.

Chapter 8 provides an in-depth view on the mining performance of the proposed LCM v.2 integration.

Conclusions and possible future work are outlined in Chapter 9.

CHAPTER 2

RELATED WORK

Research activity in data mining has initially been focused on the development of efficient in-memory data structures and algorithms to perform the complex task of frequent itemset extraction. Due to the ever-increasing capabilities of collecting and storing raw data, frequent itemset mining algorithms are now likely to deal with datasets that can easily include millions, or even billions, of transactions, sizing hundreds of gigabytes.

The first part of this chapter will briefly outline some state-of-the-art in-memory algorithms which mark significant advances with respect to the baseline techniques (e.g., Apriori). In the second part of the chapter, a more in-depth view on existing attempts to mine large datasets is given, pointing out their advantages and weaknesses.

2.1 In-memory data mining algorithms

2.1.1 FP-growth

The FP-growth approach exploits a frequent pattern tree, called an FP-tree, and an efficient FP-tree-based mining technique, FP-growth, to extract the complete set of frequent patterns by pattern fragment growth.

As described in (1), FP-growth owes its efficiency to three main points:

FP-tree compact data structure. An innovative and compact data structure, called Frequent Pattern tree, or FP-tree for short, is constructed from a transactional dataset in

the form of an extended prefix-tree, storing quantitative information about frequent patterns. Nodes in the tree represent length-1 frequent itemsets and, thanks to a clever pre-processing of each transaction, more frequently occurring items have better chances of sharing nodes than less frequently occurring ones. This results in a much higher degree of compaction in comparison with storing the dataset in transaction form.

FP-growth algorithm. An FP-tree-based pattern fragment growth mining method is adopted, which starts from a frequent length-1 pattern as an initial suffix pattern, examines its conditional pattern base (i.e., the projection database which consists of the set of frequent items co-occurring with the suffix pattern), constructs its conditional FP-tree and performs mining recursively on that tree. The pattern growth is achieved via concatenation of the suffix pattern with the new ones generated from conditional FP-trees. Unlike Apriori, which generates a large number of candidate frequent itemsets and has to perform the costly operation of pattern matching on the original dataset throughout multiple scans, FP-growth only performs count accumulation and prefix path count adjustment, which are far less expensive.

Partitioning-based search technique. The search technique employed in mining is a partitioning-based, divide-and-conquer one, rather than an Apriori-like bottom-up generation of frequent itemset combinations. This dramatically reduces the size of the conditional pattern base generated at the subsequent level of search as well as the size of the corresponding conditional FP-tree. Furthermore, the problem of finding long frequent patterns is reduced to looking for shorter ones and then concatenating the suffix.

All these techniques contribute to substantial reduction of search costs and enable the FP-growth approach to successfully tackle the mining process of much larger datasets than it is possible with Apriori.

2.1.2 NonordFP

The NonordFP approach (2) is based on a variation of the FP-tree structure employed by FP-growth. Compared to the latter, NonordFP exploits a more compact representation that allows faster allocation, traversal and, optionally, projection. It maintains less administrative information (e.g., nodes do not need to store their labels, no header lists and children mappings are required, only counters and parent pointers) and allows more recursive steps to be carried out on the same data structure, without the need to rebuild it as it is the case with FP-growth.

Drawbacks of never rebuilding the tree also exist: although projection is possible to filter conditionally infrequent items, the order of the items cannot be changed to adapt to the conditional frequencies, and hence the acronym of the algorithm.

In addition to the theoretical innovations that lay the foundations of this algorithm and its data structures, NonordFP presents a thorough handling of implementation issues, memory layout, I/O acceleration and library functions and exhibits better performance than FP-growth in most cases.

2.1.3 LCM v.2

LCM v.2 is an efficient itemset mining algorithm that currently outperforms any other existing approach, thanks to some clever data structures. Different versions of LCM are able to solve different but related frequent itemset mining problems, by finding all frequent itemsets, all

frequent closed itemsets and all frequent maximal itemsets. This is particularly advantageous when large datasets and low support thresholds are concerned, because the number of frequent patterns can easily become huge and the results of the mining process unmanageable. In this case, alternative (but complete) representations of frequent itemsets, such as closed frequent itemsets, can be highly beneficial.

As outlined in (3), LCM relies on two efficient techniques, prefix preserving closure and occurrence deliver, whose time complexity is theoretically bounded by a linear function in the number of frequent closed itemsets, unlike all other existing algorithms. Moreover, the framework of LCM is simple and needs no sophisticated data structures such as binary trees: LCM is in fact implemented with arrays only. These array-based structures are populated during an additional reading of the in-memory dataset representation and are accessed during the mining phase, speeding up the frequency counting.

As a main drawback, these structures need to be fully memory-resident to avoid vanishing the huge benefit they produce. They cannot be lazy-loaded, since they are expected to work as associative arrays and provide random access in O(1) time. Nonetheless, some optimizations are possible, as discussed later in Section 6.3.

2.2 Dealing with large datasets

As previously outlined, all the best performing frequent itemset mining algorithms heavily rely on efficient memory-resident data structures. In addition, most of them scale non-linearly with the size of the data they have to process. In such a context, as soon as these algorithms are

applied to larger datasets, critical issues arise such as extremely long execution times, memory footprint exceeding the size of main memory, massive I/O required, and a huge search space.

In the last years the problem of large dataset mining has received increasing attention and a number of solutions have been investigated. While some of them have been focused on the optimization of already existent in-memory algorithms, others have tackled the problem from different and sometimes innovative viewpoints. The following sections will present the main approaches that have been explored in the literature.

2.2.1 Optimizing existing in-memory algorithms

This class of approaches aims at optimizing the data structures and/or some procedures of already existent in-memory algorithms. They sometimes introduce novel and more compact structures which reduce main memory occupation and therefore push the scalability limit a little farther. For this reason, they can hardly be considered as real solutions, but they are nonetheless interesting because the devised techniques can sometimes be profitably applied to other classes of solutions.

An interesting proposal, COFI-tree (Co-Occurrence Frequent Item-tree) mining (4), was presented by M. El-Hajj and O. R. Zaïane. The idea behind this approach is a new anti-monotone property called the global frequent/local non-frequent property, which extends the Apriori principle, because it is able to eliminate items belonging to the i-itemset which are sure not to participate in the (i+1) candidate set. It exploits an ad-hoc tree structure, called the COFI-tree.

The algorithm for building the COFI-trees can be broken down into two phases. In the first phase an FP-tree with bidirectional pointers is constructed in main memory. In the second phase COFI-trees are built for each frequent item, one item at a time, starting from the least frequent one. The mining phase occurs independently for each COFI-tree, which is then discarded before the next one is built.

The main accomplishment of this approach is the significant amount of memory it saves by comparison with FP-growth (one order of magnitude less). This is achieved by means of the non-recursive technique used in the mining process and the pruning method exploited to remove all local non-frequent items. It also, sometimes considerably, outperforms FP-growth in terms of speed.

Another innovative approach is the UT-mine algorithm proposed by Fei-Yue Ye et al. (5). It is explicitly targeted at sparse databases, which can be efficiently represented in main memory by means of a new data structure, the Unit Triplet (UT). These data structures are populated once during an initial scan of the dataset and then read multiple times to produce k-itemsets. The main advantage of this approach is the reduced amount of memory and dataset scans it necessitates, compared to FP-growth.

2.2.2 Partitioning the original dataset

Solutions that mine large datasets by partitioning are quite straightforward to implement and are often the preferred choice. In principle, this technique could be applied to any in-memory algorithm, allowing it to deal with very large datasets. It is sufficient to:

1. split the original dataset so that each partition fits in main memory;

2. mine these partitions by means of the mining algorithm of choice;

3. join the results performing a union on the itemsets found for each partition.

Simple as it looks, this solution hides two significant disadvantages, namely the cost of the joining phase and the waste of CPU time due the over-computation generated by a non-pruned search space. Many useless itemsets are computed in each partition and discarded during the joining phase, being only locally frequent.

The widespread Partition (6) algorithm follows this principle, based on a divide-and-conquer strategy, but suffers from the aforementioned limitations. The latter are partly overcome by algorithms such as H-Mine (7), which finds globally frequent items before partitioning and can thus prune each partition of all infrequent items. In addition, it introduces an improved joining method.

The UT-mine algorithm described in the previous section can also resort to partitioning. In the case where the Unit Triplet structures cannot be accommodated in main memory, the transaction database is split and itemsets are identified in each partition. During the join phase, four cases may occur:

- an itemset satisfies the local support threshold in every partition, so it is global frequent;
- an itemset satisfies the local support threshold in some partitions and the accumulative support across those partitions satisfies the global support threshold, so it is global frequent;

- an itemset does not satisfy the local support threshold in any of the partitions, so it is global infrequent;
- an itemset satisfies the local support threshold in some partitions, but the accumulative support across those partitions does not satisfy the global support threshold; in this case, a further scan of the remaining partitions is required to compute the local supports, which are then summed to obtain the global support and determine if the itemset is global frequent.

On the whole, when large-sized databases are involved, at most two complete scans of the dataset are required by this algorithm. Even so, it is hardly scalable on huge datasets.

2.2.3 Exploiting relational DB structures

A different class of approaches is that of SQL-based data mining. The intuition that existing DBMS frameworks could be profitably exploited to perform a number of data mining tasks is the driving force of this family of solutions. The bottom-line is "not to reinvent the wheel": DBMSs already offer optimized disk access primitives and intrinsically handle main memory shortcomings. As a consequence, turning the data structure into a relational form allows the exploitation of these efficient, ready-made techniques and could in principle yield excellent mining performance on large datasets.

The first proposed SQL-based algorithm described in the literature is the SETM algorithm (8). The principal problem with this algorithm is the large number of join operations yielding the candidate set, which is computed prior to the actual mining and support counting phase. For huge datasets, this approach becomes unfeasible.

In the recent times, solutions that try to integrate efficient mining algorithms directly in the DBMS have started to appear. DBFP-growth (9) proposes a novel, disk-based FP-tree representation in the form of relational DB tables. In order to perform the mining task, the proposed approach implements the FP-growth algorithm, whose highlight is that of not generating any candidate set. The implementation resorts to Oracle PL/SQL stored procedures. The advantage over an SQL-based solution is that stored procedures are compiled separately and stored permanently in the database, rather than being processed each time, one SQL statement at a time. This yields better performance, while still providing all the functionality of SQL.

A third approach is the one adopted by I-Mine (10): a covering index, integrated in PostgreSQL, supporting, in principle, any kind of mining algorithm. I-Mine provides an exhaustive representation of the original dataset by means of an indexed FP-tree-based structure, stored in relational form.

2.2.4 Relying on virtual memory

For over two decades, operating systems have provided programmers with a virtual address space that is significantly larger than the physical one. Paging mechanisms are responsible for moving data and instructions in and out of main memory, as and when needed. Commodity PCs traditionally shipped with 32-bit processors, which, regardless of the actual physical memory size, limited the maximum memory address space to 4 GB. Nowadays, desktop PCs are typically afforded 64-bit CPUs, which translates to a nearly unlimited amount of virtual memory, bounded only by the available disk space.

As a consequence, in-core mining algorithms could in principle exploit this large amount of virtual memory to process out-of-core datasets, while relying on the operating system's paging mechanism to automatically handle data movement between main memory and secondary storage in a transparent way. The key benefit of such an approach is its implementation simplicity, since no change is required to the original in-memory algorithm, which can operate just as if its entire address space were physically available.

In practice, however, as soon as the process runs out of physical memory and starts swapping to and from disk, the CPU becomes largely idle, which renders virtual memory-based solutions extremely inefficient. This is especially true of data structures and algorithms which exhibit poor spatial and temporal locality, forcing continuous swapping and causing the processor to wait on the completion of a page fault for the overwhelming majority of time.

In the worst case even the available virtual memory may not suffice, making the mining process abort.

2.2.5 Exploiting disk-based structures

Relying solely on main memory is no longer an option when facing large datasets and alternative solutions are quickly becoming imperative. Approaches that explicitly target the exploitation of efficient disk-based data structures have also been proposed to support the extraction of knowledge from large datasets.

Ramesh et al. (11) proposed B+tree-based indices to access data stored by means of either a vertical (e.g., ECLAT-based) or a horizontal (e.g., Apriori-based) representation. However, these solutions are usually worse than, or at best comparable to, flat file mining.

El-Hajj and Zaïane (12) proposed a disk-based data structure, called Inverted Matrix, to store the transactional database in an in inverted matrix layout. The COFI-tree algorithm is then exploited for frequent itemset extraction. This solution is still not widely applicable, because it specifically addresses very sparse datasets, characterized by a large number of items with unit support.

Grahne and Zhu (13) proposed the Diskmine approach, which uses recursive projections to partition the data until it fits in main memory and subsequently exploits an in-core algorithm (FP-growth) to mine it. Projected partitions are written on disk and retrieved as needed. At the end of the mining process, the frequent itemsets can be computed by taking the union of the itemsets mined from each projection. The major downsides to this approach are the several, costly accesses to the potentially large number of materialized partitions and the significant disk space the latter may require. In addition, when the frequent 1-itemset projection does not fit in memory, then all combinations of frequent 2-itemsets must be projected, even if a large subset of these itemsets are not actually frequent. One of the main advantages of FP-growth (i.e., no need for candidate generation) is thus eliminated.

2.2.6 Directly managing the I/O

Reducing the computational complexity of mining algorithms and the main memory requirements cannot, by itself, yield true scalability. On the other hand, when relying on secondary storage, the vast majority of time is spent with I/O operations; approaches that claim to tackle huge dataset mining in a timely and scalable fashion cannot disregard an efficient I/O manage-

ment. This can be achieved by exploiting an already available I/O framework, such as the one of a DBMS, or by developing I/O-conscious applications.

A significant example of direct I/O management that has led to optimal results in terms of execution times, CPU utilization and memory footprint has been provided by Buehrer et al. (14). They proposed several I/O conscious optimizations to construct and subsequently mine a disk-resident FP-tree structure which exploits a clever physical organization and an ad-hoc data access infrastructure.

The creation process benefits from a preliminary partitioning and approximate hash sorting step, which enables to build a prefix tree on disk while keeping in main memory only the currently elaborating chunk. This technique dramatically reduces the number of page faults and the overall creation time.

Once created, this tree is reallocated in virtual memory by means of a depth-first visit: this ensures spatial locality during the mining phase, which traverses the prefix tree in a bottom-up fashion. The mining process is additionally restructured to exploit temporal locality by maximizing the reuse of the prefix tree once it is fetched into main memory. This is accomplished by breaking down the tree into blocks of memory along paths of the tree from leaf nodes to the root. Each block is iteratively loaded in main memory and the conditional pattern bases for all items that hit the block are computed at once.

Another efficient I/O conscious disk-based solution has been proposed by M. Adnan and R. Alhajj under the name of DRFP-tree (Disk-Resident Frequent Pattern-tree) (15). This novel approach initially constructs an FP-tree in main memory and tries to mine it with the

traditional FP-growth algorithm; it will then expand into secondary storage, creating a disk-resident FP-tree and mining it with its own disk-resident version of FP-growth, only if it runs out of physical memory.

During a pre-processing phase an exact sorting is performed on the original dataset. After that, a memory-based FP-tree is constructed. If at some point it becomes evident that the main tree, or the projection ones needed for the subsequent mining operations, cannot be accommodated in physical memory, stale subtrees are saved on secondary storage, following a DFS order, in a compact form. This is possible because the preliminary sorting guarantees that new branches are always inserted in the rightmost part of the prefix-tree.

Experimental results show non-negligible performance degradation when the algorithm has to switch from the memory-based to the disk-based implementation. However, in-memory FP-growth fails to mine large datasets when low support thresholds are used, while the disk-based version still succeeds in the task.

CHAPTER 3

THE LSD FRAMEWORK

The proposed disk-based data mining framework, LSD (a Large-Scale Data-mining framework), exploits a persistent representation including a number of distinct data structures. Its core components are the Hybrid-Tree, containing the transactional data, and the Item-Index, which complements the tree with additional, useful information. All of these structures are stored together on the same binary file.

The Hybrid-Tree provides a prefix-tree representation of the original dataset, characterized by the following features:

- each transaction is represented by a single path in the tree, from a root to a leaf;
- paths encoding different transactions can overlap when the transactions share a common prefix;
- each node represents a single item and stores its associated local support, which equals the number of all transactions sharing the node;
- items along a path are sorted by decreasing global support.

Depending on the original data distribution, this tree representation can significantly reduce space requirements with respect to a dataset stored in plain text form: the denser the input data distribution, the higher the level of compaction achieved.

The Item-Index is an auxiliary structure enabling an item-based traversal of the tree. Because our persistent representation does not address a specific mining algorithm but was conceived to be as general as possible and support, in principle, any algorithm, it is crucial that many different kinds of navigations be allowed throughout the tree. At the same time, only a few of these algorithms require an item-based navigation of the tree; for this reason, it was deemed convenient to separate the Item-Index structure from the Hybrid-Tree, so that the former can be loaded on-demand as, and only when, needed.

3.1 The Hybrid-Tree

At an abstract level, the Hybrid-Tree is based on the FP-tree structure exploited by the FP-growth mining algorithm previously described. Its physical organization, however, is different from that of the FP-tree, because it is devised in such a way as to favor a disk-based exploitation. Its main features comprise:

- bi-directional navigation;
- double layering;
- compact representation for contiguous nodes with unit support.

Table I reports a small dataset used as a running example, and Figure 1 shows the complete materialized structure of the corresponding Hybrid-Tree and Item-Index. In the header table, items have been sorted by decreasing global support. Each of them is linked to its own Item-Index chain. Item-Index entries have been depicted using different colors for upper-layer nodes (purple-violet squares) and lower-layer nodes (red-salmon squares). To avoid overcrowding the design with arrows, only some node-link pointers have been represented.

TID	Items
T1	A, B, D, E, F, G, K, L
T2	A, C, D, G
T3	E, F, H, I, O
T4	A, B, C, D, E
T5	A, B, E, F, G, M
T6	C, E, F, S

TABLE I

EXAMPLE DATASET

3.1.1 Bi-directional navigation

The Hybrid-Tree can be traversed in both a top-down and a bottom-up fashion.

The bottom-up traversal is made possible by storing in each node the pointer to its parent node.

The top-down traversal, on the other hand, was realized following a linked list approach. More specifically, each node stores the pointer to its first child node and is linked to its next brother, if any. Brother nodes can thus be reached by following the next-brother pointer, until the end of the chain. This structure is well-suited for a disk-based environment, since nodes do not need to store neither the pointers to nor the number of their children, and therefore have a constant size.

In addition, the lists of brother nodes are sorted by decreasing global support of the items. This requires a small extra computational effort, but provides a considerable benefit during the creation phase and whenever a support-based operation is being performed on the prefix-tree.

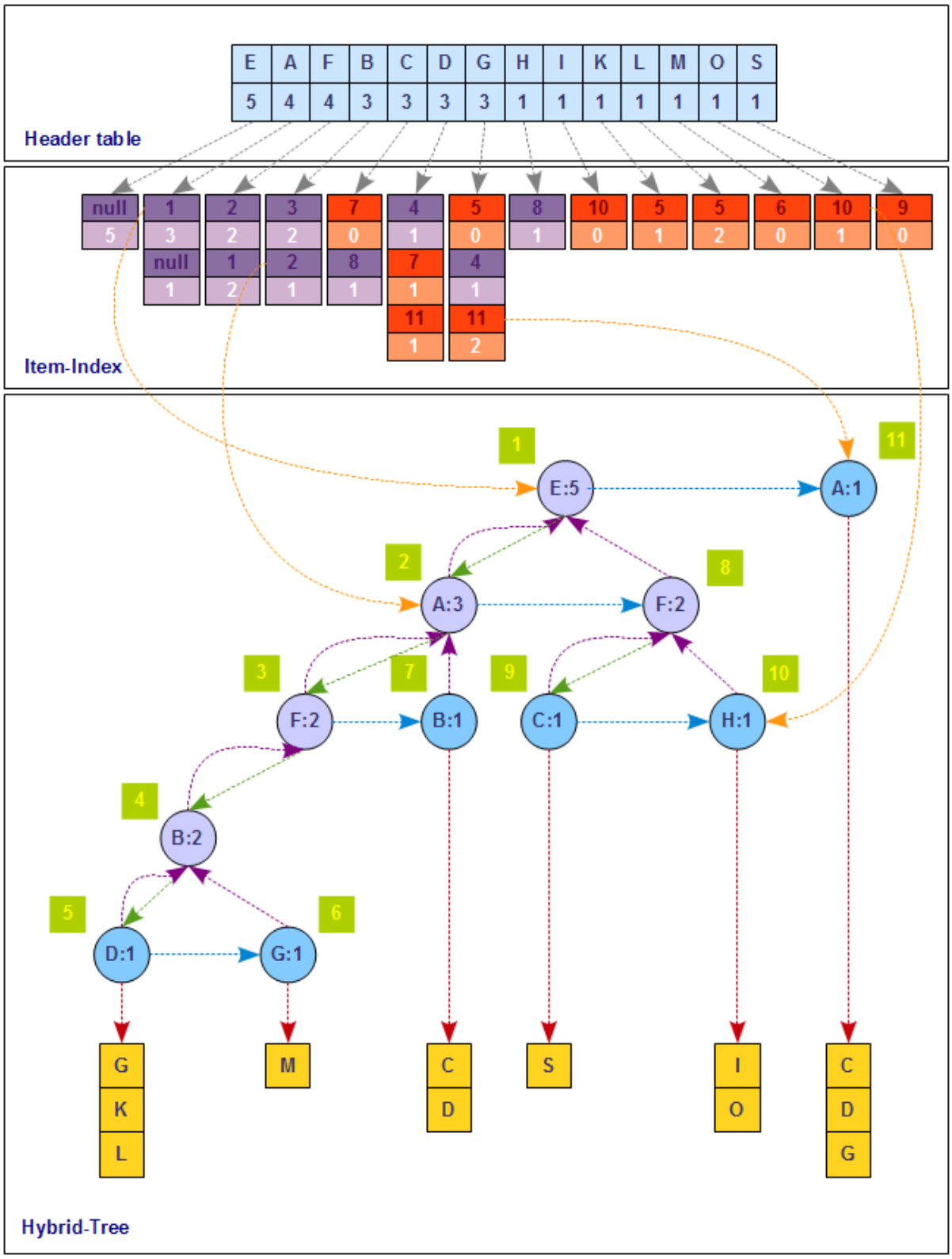

Figure 1. Materialization of an example dataset

3.1.2 Double layering

In order to provide an effective and compact representation for diverse data distributions, the Hybrid-Tree is characterized by two different node structures, which are kept in different portions of the tree, or layers: the upper layer and the lower layer. The Hybrid-Tree double-layering is illustrated in Figure 2.

Upper-layer nodes

A node is classified as belonging to the upper layer if the local support of the item it represents is strictly greater than one. In this case, the node is shared by a relatively large number of transactions and, as a consequence, it is included in the dense portion of the dataset. Being located in the upper part of the tree, such nodes correspond to items with a high global support and are thus likely to be frequently accessed during the mining process. For instance, node 1 in Figure 1 is an upper-layer node.

Upper-layer nodes are full-fledged tree-node structures comprising:

- an item ID;
- its local support;
- composite pointers to the parent, first child and next brother nodes;
- the length of the chain of unit support children; the meaning of this field will become clear in the following.

The upper node structure is depicted in Figure 3.

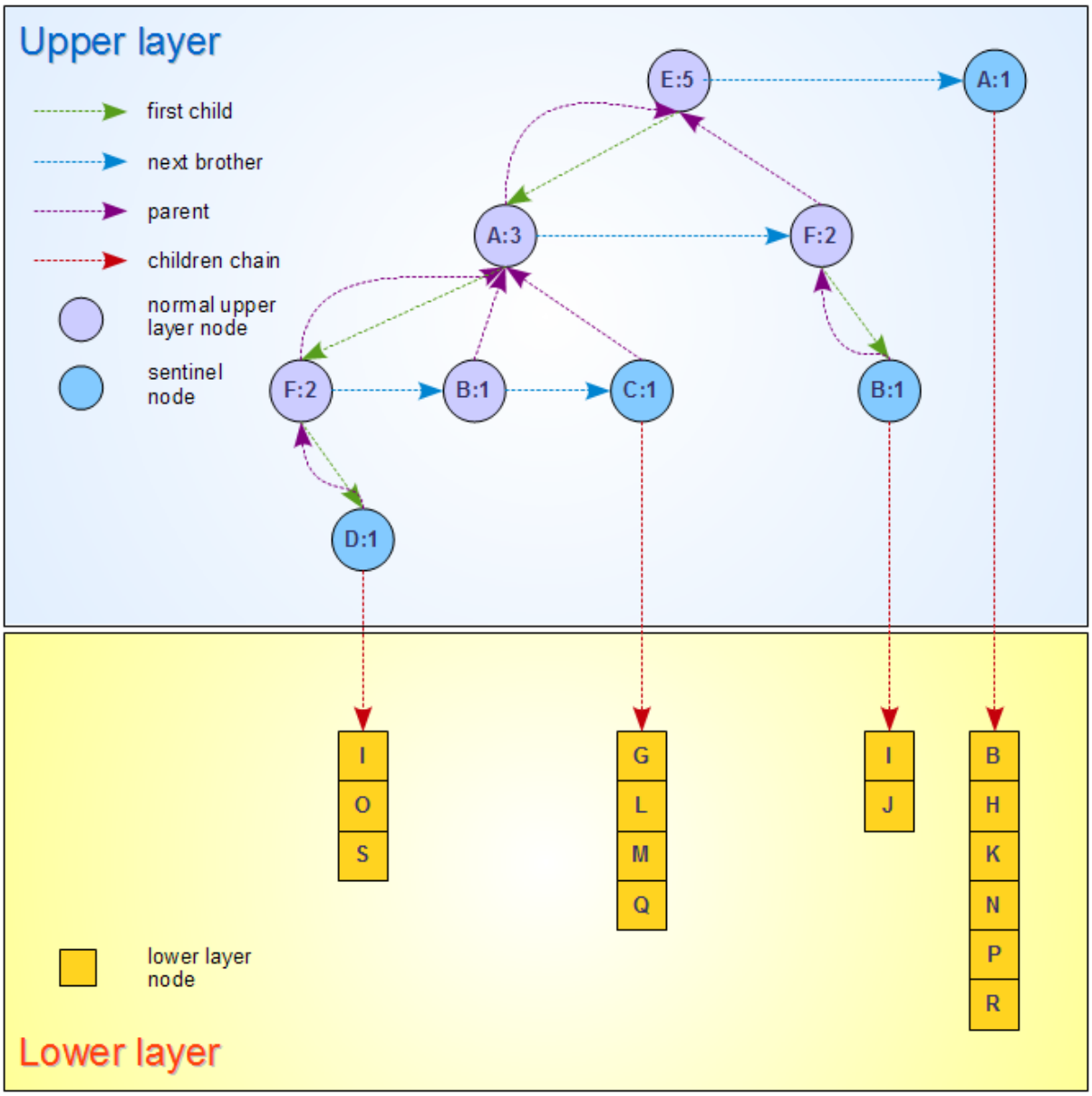

Figure 2. The Hybrid-Tree layering

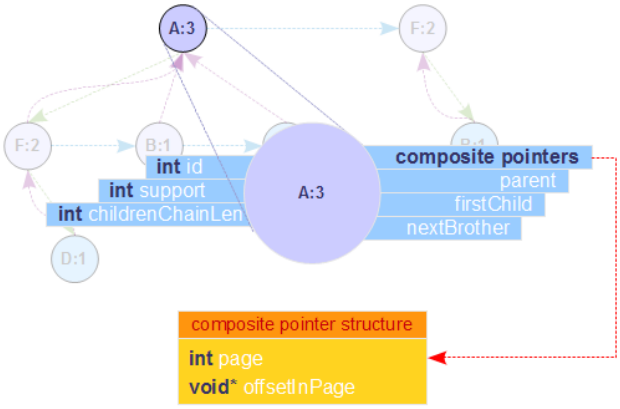

Figure 3. Upper-layer node structure

Lower-layer nodes

The lower layer includes nodes whose local support is unitary. These nodes can have at most one child, because they are not shared by multiple transactions, and hence a sub-path starting from a unit support node is a chain of unit support nodes with no branches. As with upper-layer nodes, the physical location of lower-layer nodes within the tree suggests that these nodes typically associate with items with a low global support. For this reason, lower-layer nodes are only read few times during the mining process. Lower-layer nodes are depicted in yellow in Figure 1.

Yet, if the dataset is very sparse, the physical organization of these nodes on disk may significantly impact performances in the mining phase. When these chains of unit support nodes make up the majority of the tree, it is essential to group them together in the same layer.

All lower-layer nodes having a local support equal to one and a single child, it is possible to devise a smaller structure to represent them than it was used for upper-layer nodes. The candidate structure shall be able to include:

- item ID;
- parent-child relationship.

There is no need to store the local support of the item, since it is implicitly one. As a consequence, these nodes can be represented as integers and chains of lower-layer nodes can be represented as arrays of integers; node contiguity in the sub-paths is thus given by cell contiguity in the arrays. The lower-layer node structure is depicted in Figure 4.

In order to link together sub-paths located in different layers and belonging to the same path, a special sentinel node is employed. This node is the first in the unit support chain. Hence, it is functionally a lower-layer node, but its data structure is a specialization of the upper-layer node structure. In particular, being its parent an upper-layer node, the sentinel node may have a brother node. Its unique child node is the first cell in the array representing a sub-path of lower-layer nodes. The length of this array is stored in the `childrenChainLen` field of the upper-layer node structure. For example, node 5 in Figure 1 is a sentinel node.

The latter field has a twofold meaning: on the one hand, it allows distinguishing sentinel nodes (non-zero value) from normal upper-layer nodes (zero value); on the other hand, it is used when a sentinel node is encountered to fetch the corresponding sub-path array by means of a single disk read.

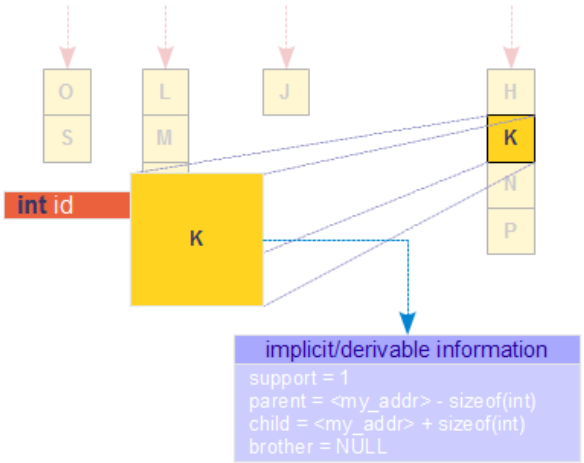

Figure 4. Lower-layer node structure

While this technique is adequate for a top-down traversal, navigating these sub-paths in a bottom-up fashion still seems critical. Loading nodes backwards one at a time from disk, until the start of the chain is reached, would entail several (costly) read operations. Furthermore, some kind of chain delimiter and a pointer to the sentinel node would be required. It has been chosen, instead, to store the number of cells preceding each node, so that only that portion of the array can be fetched at once. To avoid increasing the lower-layer node size, this information is inserted in the Item-Index structure.

3.2 The Item-Index

The Item-Index is an additional structure providing support for item-based traversals. It contains the node-link chains for each item in the Hybrid-Tree and is organized as a list of arrays, each one containing as many entries as the number of tree nodes labeled with that item.

Traditional FP-trees usually embed this information directly in the tree nodes, but we decided to keep this structure separate from the Hybrid-Tree, so that only the data structures actually needed by the adopted mining algorithm can be loaded in main memory.

Every Item-Index entry consists of a structure named `NodelinkSmall` storing:

- the local support of the corresponding node, and
- the composite pointer to the node's parent.

This is to skip one disk access during the mining phase. The motivation is that some of the mining algorithms exploiting item-based projections, such as FP-growth and LCM v.2, only require the parental chain of items from the node to the root and the local support of the node, but not the node itself.

For lower-layer nodes, the above still holds with a slight variation. Since the local support of a lower-layer node is always unitary, the support field of the `NodelinkSmall` structure is used for a different purpose: the number of nodes preceding the current one in the sub-path array is stored instead of the support, as explained in Section 3.1.2. In addition, the composite pointer field contains the address to the sentinel node of the unit support chain. Using these two pieces of information, one can retrieve the portion of the chain, down to the item at stake, with a single read operation.

The entry point to the Item-Index is inside the header table of the tree. For each item, the latter contains:

- the item ID;

- the item support;
- the length of the node-link chain;
- the starting offset of the node-link chain within the materialization file.

As it was the case with unit support node chains, an entire node-link chain can be fetched at once, thus reducing the amount of I/O cycles.

CHAPTER 4

BUILDING THE HYBRID-TREE

The Hybrid-Tree relies on an optimized physical organization of data in order to provide an efficient distribution-adaptive representation and ease data retrieval in the mining process. To this aim, the most suitable data structures to store different portions of the dataset are selected based on the local support of each node, as outlined in Chapter 3.

Unfortunately, this information is only known once the entire transactional dataset has been converted into a prefix-tree. Creating a preliminary FP-tree in main memory is unfeasible when no support threshold is enforced, because the size of such a tree would, in most cases, far outstrip the available physical memory. As a consequence, this issue has been addressed by creating a temporary disk-resident tree, hereinafter called G-tree.

This tree structure is then visited to assign each node a layer and a location in the final tree. The Hybrid-Tree and the Item-Index are eventually written on disk.

4.1 The G-tree

The "growing" tree, or G-tree is a disk-resident FP-tree used as an auxiliary structure during the creation of the Hybrid-Tree. It is built according to the traditional rules used to construct an FP-tree, but its nodes are stored on disk.

It is a "growing tree" because it is incrementally built, as more transactions are added, by inserting new nodes and updating the support count of existing ones.

Once constructed, it serves two purposes:

- it enables choosing the appropriate physical structure for each node, according to its local support;
- it stores the final position of each node in the Hybrid-Tree.

The latter information is not as crucial because it says where each node shall be written in the materialization file – which could easily be done by computing its final position on-the-fly – as it is because it specifies where its children and brother nodes are located. It would otherwise be very challenging to set links between nodes within the materialized tree.

Owing to its growing nature, the G-tree tree must provide a high degree of flexibility. This is achieved by means of versatile structures such as lists (in place of arrays), at the cost of an increased disk space usage compared to the final tree.

4.2 Data pre-processing

Turning the transactional data into a prefix-tree structure would be a straightforward task, were it possible to fit it in main memory. Unfortunately, this is not the case with large-sized datasets. Dealing with the G-tree, on the other hand, raises a number of issues which typically only affect secondary storage.

Because of the large number of updates each prefix-path can go through, it would be desirable to have nodes along the selfsame path written in contiguous disk locations. However, for a given node, the number of its children, if any, is not known until they have been created, so it is impossible to reserve contiguous space for them. Similarly, nodes with a low local support

are likely to be updated only few times throughout the creation process and could be handled separately, but there is no way to know in advance whether a given node will have a high or a low support once the tree is complete. It thus appears that the optimizations exploited on the Hybrid-Tree do not hold for the construction of the intermediate tree.

In order to reduce the cost of this phase, a pre-processing step is performed prior to building the G-tree. In particular, the input dataset is arranged in such a form that the chances that the same tree path be reloaded multiple times just to increase its support are minimized.

4.2.1 Preliminary steps

As a preliminary step, the entire dataset is read once. For each transaction, newly found items are inserted in an ad-hoc structure, while the support of previously found items is updated accordingly. Items are then sorted by decreasing global support and pruned of infrequent ones. A unique and progressive ID is finally assigned to each item, in descending order of support.

By means of a second dataset scan, each transaction is read again. Items in every transaction are remapped using their new IDs and sorted by ascending order of ID. This is equivalent to a descending order of support count: in a transaction, item x precedes item y if and only if the frequency of x is greater than or equal to the frequency of y. The remapped dataset is written, transaction by transaction, on a temporary file.

Finally, transactions are sorted in lexicographic order, that is, for any two transactions T_i and T_j, $T_i < T_j$ (i.e., T_i precedes T_j in the final ordering) if and only if T_i and T_j agree on the first $k \geq 0$ items and the frequency of the $(k+1)$-th item in T_i, if any, is greater than the frequency of the $(k+1)$-th item in T_j.

Because of this total order defined on the transaction set, transactions sharing the same prefix-path are contiguous: as a consequence, only the last accessed tree path may need to be updated when a new transaction is inserted. This means that, once a transaction with a new prefix is found, the previous blocks of the tree are sure not to be used anymore, saving on I/O costs. Moreover, the first transactions to be processed are the ones containing the most frequent items, whose nodes will thus be allocated sequentially.

On the whole, transaction sorting allows a significant speed-up and produces a better organization of the G-tree on disk.

The sorting strategy employed for this task will be described in the following section.

4.3 Sorting the dataset

As previously outlined, the sorting phase plays a fundamental role in the process of creating the G-tree on disk. However, it needs to be carefully devised, lest it become the most time-consuming one.

Due once again to the large size of the datasets that we intend to materialize with this persistent representation, transactional data cannot be fully memory-resident during this task. It is therefore necessary to adopt a partitioning strategy of some sort.

4.3.1 The Linux `sort` command

In the early development stage of the disk-based framework, the Linux `sort` command was exploited to perform the initial sorting step. The `sort` utility, implemented on all POSIX systems, reads one or more text files (or the standard input) and sorts their lines in lexicographic order, or according to whatever order is specified on the command line.

Since this utility is intended to be able to deal with large files, the adopted sorting strategy is an external R-way merge sort. This means that the input file is initially split into a number of equally-sized chunks, which are sorted in main memory and written on disk. The size of the initial chunks is chosen based on the available physical memory. Chunks are then recursively merged R at a time until a single fully sorted file is obtained.

The default behavior is to sort the input file using entire lines of text as keys and following an ASCII lexicographic order, but `sort` can also be instructed to identify key fields within each line and, optionally, to treat them as numbers instead of strings. The correct way to sort transactions (according to the order described in Section 4.2.1) is by means of the following command:

```
sort remapped-file.txt -n -k1 -k2 -k3 ... > sorted-file.txt
```

The above line means that several blank-separated fields, namely items (`-k1 -k2 -k3 ...` options), are present in each line and that they should be considered as numbers (`-n` option).

Shortcomings

Although reportedly efficient in a variety of applications, the `sort` utility revealed poor performance in the specific case of transactional dataset sorting.

One possible reason for this can be identified with the large amount of disk reads that take place when the merged partitions cannot be accommodated in main memory. The merge-sort approach works well as long as all the recursive merge steps can be carried out in main memory. If this is not the case, the algorithm starts swapping to disk. Since multiple steps are required to merge all the (increasingly large) chunks into a sorted file, this translates to portions of the

dataset being continuously read from and stored on disk, which negatively impacts execution times. With very large datasets, the majority of time is spent with I/O operations and this approach becomes unfeasible.

A further cause of slow-down is the fact that the algorithm operates on text files. Data to be sorted and merged is always stored in the form of strings and, every time a comparison is made, text-to-number conversion must be explicitly carried out. This results in a relevant fraction of the CPU effort being spent only with data-type conversions and, eventually, in long execution times.

4.3.2 Proposed sorting algorithm

The general idea behind the proposed sorting algorithm relies on the divide and conquer algorithmic paradigm again. Its major accomplishment is that of splitting the sorting step into a number of independent sorting tasks, so that no merging is ever required, thus saving on I/O costs.

The adopted strategy partitions the dataset into disjoint chunks, defining a partial order on the set of transactions and a total order on the set of chunks.

More in detail, given any two chunks C_i and C_j such that $C_i < C_j$ (i.e., chunk C_i precedes chunk C_j in the total order defined on the set of chunks), any transaction T_{i_k} belonging to chunk C_i precedes any transaction T_{j_k} belonging to chunk C_j in the partial order defined on the set of transactions, and hence also in the total order.

In order to split the dataset into chunks, an initial scan is performed to count the number of transactions starting with each item. Sets of transactions starting with the same item will be hereinafter referred to as "first-item projections".

The contents of each chunk (i.e., the set of first-item projections it includes) are then computed by scanning the counts obtained in the previous step. First-item projections are assigned to the current chunk, while contextually updating the number of transactions currently contained in the chunk, until it is full. This is done in such a way that no first-item projection ever spans across multiple chunks, so as not to violate the partial order previously discussed.

The dataset is then read a second time and transactions are inserted in their assigned chunks, stored as separate files on disk. Chunks are then iteratively fetched from disk into memory, sorted independently of each other using the quicksort algorithm and written back to disk in sequence. Thanks to the aforementioned property, the resulting file will be completely sorted. A brief outline of this procedure is provided in Algorithm 1.

Time complexity

Besides reducing the time required to perform the sorting step by several orders of magnitude, this strategy scales very well with large datasets, because it has a linear complexity in their size.

Let us consider a dataset D containing n transactions. Let us also assume that fixed-size partitions containing p transactions each are being employed.

Dataset scans have a cost proportional to its size; we may assume different proportionality constants for the two scans, since the dataset is read in the one case (with unit cost k_1) and read and written at the same time in the other case (with unit cost k_2).

The cost of computing the contents of each chunk depends on the number of different items occurring in the first position; this value is upper-bounded by the number of distinct items in the dataset and does not depend on the dataset size, so it can be considered constant.

Finally, the quicksort algorithm has a $p \log p$ complexity, p being the fixed partition size, and it is applied to every partition, which needs to be loaded from (with unit cost k_3) and stored back (with unit cost k_4) to disk. Summing up:

$$C(n) = k_1 n + k_2 + k_3 n + \frac{n}{p}(k_4 + p \log p) = \Theta(n)$$

Implementation notes

Some implementation details that turned out beneficial are explained hereinafter.

One of the differences with respect to the `sort` utility is that the proposed algorithm operates directly on binary data. This is the format used to represent and process transactions inside the disk-based framework and it is the most natural one, since each transaction can be simply stored in array form. Consequently, the remapping step (described in Section 4.2.1) has been modified so as to produce a binary file instead of a text one. Reading such a file is faster, because no character parsing is required. In addition, because the output sorted file is binary, I/O and processing cost reduction also affects the subsequent G-tree creation phase.

The remapping now also integrates the preliminary step of the sorting algorithm, namely the first-item occurrence counting. This is obtained at virtually no cost, but permits one less dataset scan.

Finally, in order to minimize the number of disk accesses during the dataset splitting phase, buffers have been directly handled, without resorting to the buffering mechanism provided by the standard C library I/O functions. This is particularly critical with very large files: since many partitions are produced, continuous shifts from one to another to write small amounts of data would dramatically impact performance.

4.4 Building the G-tree

4.4.1 Chunk-tree

The pre-processing step provides a considerable benefit to the creation of the disk-based G-tree. In order to further improve this process, another module is chained to the data pre-processor.

To reduce the I/O cost, the sorted transactional dataset is split again into equally-sized chunks. One chunk at a time is considered and a temporary FP-tree, named a chunk-tree, is built in main memory. The chunk-tree is then merged with the current G-tree on disk and is finally discarded.

The merging step is carried out by visiting in a depth-first order both the disk-resident tree and the chunk-tree at the same time. The I/O reduction is due to the fact that each node of the G-tree is read at most once during the merge of a single chunk-tree, to update its support count or create new children or brother nodes. These operations correspond to the processing of

Algorithm 1 Dataset sorting algorithm

```
procedure DatasetSort (D: database;
                       I: items;
                       S: sorted database) is

constant partitionSize;

begin
  -- count first item occurrences
  for item i in I do
    frequency[i] := 0;
  end for;
  for transaction t in D do
    frequency[t.firstItem] := frequency[t.firstItem] + 1;
  end for;

  -- compute target partition for each first-item
  numPartitions := D.size / partitionSize;
  partitions := new Partition[numPartitions];
  for n in 1..numPartitions do
    partitions[n].available := partitionSize;
  end for;
  current := 1;
  for item i in I do
    if frequency[i] > partitions[current].available then
      current := current + 1;
    end if;
    partitions[current].available :=
          partitions[current].available - frequency[i];
    itemGoesTo[i] := current;
  end for;

  -- split dataset
  for transaction t in D do
    targetPart := itemGoesTo[t.firstItem];
    partitions[targetPart].writeOnFile (t);
  end for;

  -- sort partitions
  for n in 1..numPartitions
    chunk := partitions[n].readFromFile ();
    sort (chunk);
    S.writeOnFile (chunk);
  end for;
end DatasetSort;
```

multiple transactions which, without the intermediate chunk-tree, would have required several I/O cycles.

The preliminary sorting of the transaction set maximizes the chances that transactions within a given chunk share many prefix-paths, and hence the compression achieved by the chunk-tree, thus minimizing the number of updates on the G-tree nodes. Furthermore, at most a single chunk-tree path can overlap with one existing temporary tree path. This occurs when transactions sharing the same prefix are at the border between two consecutive chunks. This way, only a very limited portion of the G-tree is rewritten during the creation process.

4.4.2 Very Infrequent Speculation

In addition to the processing described so far, a special technique has been employed to reduce the interleaving of infrequent nodes with the frequent ones, relying on the assumption that high-support nodes are more prone to be accessed than low-support ones.

This technique mimics the one adopted to distinguish between upper-layer nodes and lower-layer nodes in the final Hybrid-Tree, but it exploits a different metric: since the local node support is not yet available, an estimate is performed based on the global item support. This value is known as soon as the header table is computed. Item supports are thus compared to a user-specified percentage of the average item support. Nodes whose item support is lower than this threshold are associated to infrequent items and are stored in the last regions of the disk space used during the creation phase, so as to keep them separate from the high-support ones. The advantage is that allegedly frequently accessed nodes are stored in adjoining disk locations, reducing I/O times.

4.5 Building the final Hybrid-Tree

Once the process of creating the G-tree on disk is complete, a number of depth-first visits are performed to retrieve statistical information, select the appropriate structure for each node and write the final Hybrid-Tree in the materialization file along with the Item-Index.

4.5.1 First G-tree visit

During the first G-tree visit, a layer of the final tree is chosen for each node. Nodes whose parent node has a unitary support are assigned to the lower layer, while the remaining nodes are assigned to the upper layer. This ensures that sentinel nodes are inserted in the upper layer. A count is kept for each type of node and it is used at the end of the process to compute an optimal size for the materialized file page.

This step is aimed at minimizing the amount of disk space that is wasted in each page, especially the last one. Because of the memory mapped I/O mechanism exploited to load portions of the tree in main memory during the mining phase, the tree page size is required to be a multiple of the OS memory system page. In addition, all pages need to be entirely written on disk, regardless of the actual number of nodes they contain, thus possibly wasting large amounts of space in the last page. As a consequence, tuning the page size to best fit the actual space requirements can reduce the materialized file size. This is particularly evident with small- and medium-sized datasets, where the orders of magnitude of used and wasted disk space are comparable.

The actual tuning is performed by means of a simple minimum-search algorithm: possible page size values are explored in sequence and for each of them the corresponding disk space waste is computed. The page size yielding the minimal waste is then selected.

4.5.2 Second G-tree visit

In the second G-tree visit each path is traversed in depth-first order again. Nodes belonging to the upper layer are assigned a position in the final materialization; this information is stored inside the G-tree nodes and used later during the write phase. Instead, lower-layer nodes are directly written on disk, since they do not need to store child, parent or brother pointers.

At the same time, node-link chains are created for either type of nodes and stored in a temporary file on disk.

4.5.3 Item-Index optimization

The node-link chain structures obtained after the second tree navigation are not very well-suited to be directly integrated in the materialization, mainly owing to the large amount of disk space they take up. As each node-link entry needs to store the node support and the composite pointer to its parent, this structure has an overall size of 12 bytes. Although smaller in size than an upper-layer node (which takes 28 bytes), a node-link entry is bigger than a lower-layer node, which occupies only 4 bytes. Since the latter type of nodes usually makes up the vast majority of the tree, this results in the node-link chains being larger than the Hybrid-Tree itself. Therefore, some techniques to reduce their size have been designed and implemented:

- a differential coding scheme to represent Hybrid-Tree page references;
- "virtual" Item-Index pages larger than Hybrid-Tree pages;

- normalized bit-field pointers to maximize packing of information.

Differential coding

The first optimization concerns the composite-pointer page field. In order to reduce the number of bits required to represent this value, a differential page numbering scheme has been adopted. In other words, the page field does not contain a page number: the difference with respect to the page number of the previous entry is stored instead. The assumption is that this difference can be represented on a smaller amount of bits than the page number itself.

When the chain is retrieved from the Item-Index in the mining phase, absolute page numbers are computed back by adding the stored values in sequence.

A potential downside to this approach is that, in order to obtain the page number of the n-th entry, it is necessary that all $n - 1$ previous values be computed as well. However, since node-link chains are always read in whole and following their sequential order, this is not a real disadvantage.

During the node-link optimization phase, each chain is preliminarly fetched from disk and sorted by increasing page number. As a matter of fact, the temporary chains are obtained by means of a depth-first visit, and the same order is followed when allocating nodes in the final tree; however, since parent pointers, and not pointers to the nodes themselves, are stored in the chain, this kind of visit cannot always guarantee the increasing order of page numbers – although this holds in the majority of cases. A sorting step is therefore required to ensure that adjacent node-link cells have increasing page numbers.

After that, each node-link chain is processed one cell at a time and encoded according to the differential scheme previously described.

This technique slightly increases the computational effort required to read and, especially, write the node-link chains (because of the sorting operation), but it allows halving the size of the page field (from 16 bits to 8 bits).

Virtual Item-Index pages

The differential coding scheme just described has a shortcoming. In particular, because the difference with respect to the previous page number is represented on 8 bits, the maximum gap between two adjacent node-link cells can be at most 254 pages (value 255 is reserved to represent the `UNDEFINED_PAGE` value).

The entity of this limitation depends on the size chosen for the Hybrid-Tree pages. Since in practice it is not convenient for the mining process to choose very large pages, this limit can become substantially low. When dealing with large datasets, nodes labelled with the same item can be spread across several gigabytes and their distance may exceed the maximum value that can be represented on 8 bits. In such cases, the differential scheme would fail to compress the node-link chain.

In order to widen the capabilities of the differential coding technique, a "virtual" paging scheme has been introduced in the Item-Index. As a consequence, the page sizes in the Hybrid-Tree and in the Item-Index no longer need to be identical. The Hybrid-Tree page size can thus be tweaked to yield the best performance in the mining phase, while a larger Item-Index virtual page size may be chosen so as to enable the differential scheme to deal with node-link chains

on large datasets. When creating the Item-Index, composite pointers referring to small-sized tree pages are remapped to the larger index pages: multiple tree pages are thus represented in the Item-Index by means of a single page, thus extending the maximum distance between two contiguous nodes.

An example will clarify this concept. Let us assume that pages in the Hybrid-Tree have a size equal to S_{HT} bytes. If we are to use the same page size within the Item-Index and exploit the differential coding scheme, the gap between the physical locations of the nodes referenced by two contiguous node-link cells is allowed to assume values in $]0, 255 \cdot S_{HT}[$ bytes. On the other hand, exploiting the virtual paging scheme with an Item-Index virtual page size equal to $S_{II} = kS_{HT}, k \in \mathbb{Z}_0^+$, such gap is now in range $]0, 255k \cdot S_{HT}[$ bytes, which is k times larger than before.

In order to reduce the number of multiplications and divisions, k is chosen to be a power of 2, so that such operations can be performed by means of bit shifts.

Normalized bit-field pointers

A further optimization has been performed on the composite-pointer offset field. In particular, this field is used to carry the relative address of parent nodes within their corresponding page. Since the referenced parent node is an upper-layer node for upper-layer nodes and a sentinel node (i.e., still an upper-layer node) for lower-layer nodes, this value is always a multiple of the upper-layer node size. It is thus possible to normalize it with respect to this size, so as to reduce the number of bits needed to store it.

As a consequence, real pointer values are normalized with respect to the upper-layer node size during the Item-Index creation phase, and the reverse step is performed upon decoding Item-Index chains. The normalized offset value can thus be represented on 24 bits instead of 32. A data type with such a bit-width is somewhat unusual and is not provided by the C/C++ programming languages. It has been implemented as a combination of 16-bit and 8-bit data types. Furthermore, to avoid shift and/or bit-mask operations, a helper-pointer type has been created, resorting to a C `union`: the normalized offset value is thus assigned to an auxiliary helper-pointer variable and its low- and high-order parts are retrieved by simply accessing ad-hoc `union` fields.

Thanks to the combination of these techniques, the Item-Index can be eventually compressed by 33%; in addition, because both the creation process and the mining phase are largely I/O-dominated, none of the operations required by the compression and decompression steps noticeably impacts performance.

4.5.4 Hybrid-Tree serialization

As the last step, the disk-resident G-tree is traversed a third time in a depth-first order. Nodes along each prefix path are written on disk in the contiguous locations assigned during the second traversal. Links to parent, first child and next brother nodes are established by accessing the corresponding G-tree nodes and retrieving their allotted positions in the final tree.

4.6 Materialization format

The final materialization is stored on a single binary file, comprising a number of different structures:

- a materialization descriptor, summarizing the features of the tree;
- a header table for the prefix tree, containing a list of all items with their support and entry point to the Item-Index;
- a remapping table;
- the Item-Index;
- the Hybrid-Tree.

The format of the materialized representation is depicted in Figure 5.

The Hybrid-Tree structure is further divided into pages, stored sequentially in the materialization file. This provides fine-grained access to the tree, because each page can be independently fetched from disk and loaded into main memory by exploiting the BufferCache data access framework, as explained in Section 5.2.

4.6.1 Physical organization

The physical organization of the Hybrid-Tree on disk has been designed to reduce the number of disk reads performed during data retrieval.

Support-based projections are commonly exploited by a number of frequent itemset mining algorithms. In order to extract such projections from a prefix-tree representation, a top-down depth-first visit needs to be performed. Item-based traversals, made possible by the Item-Index,

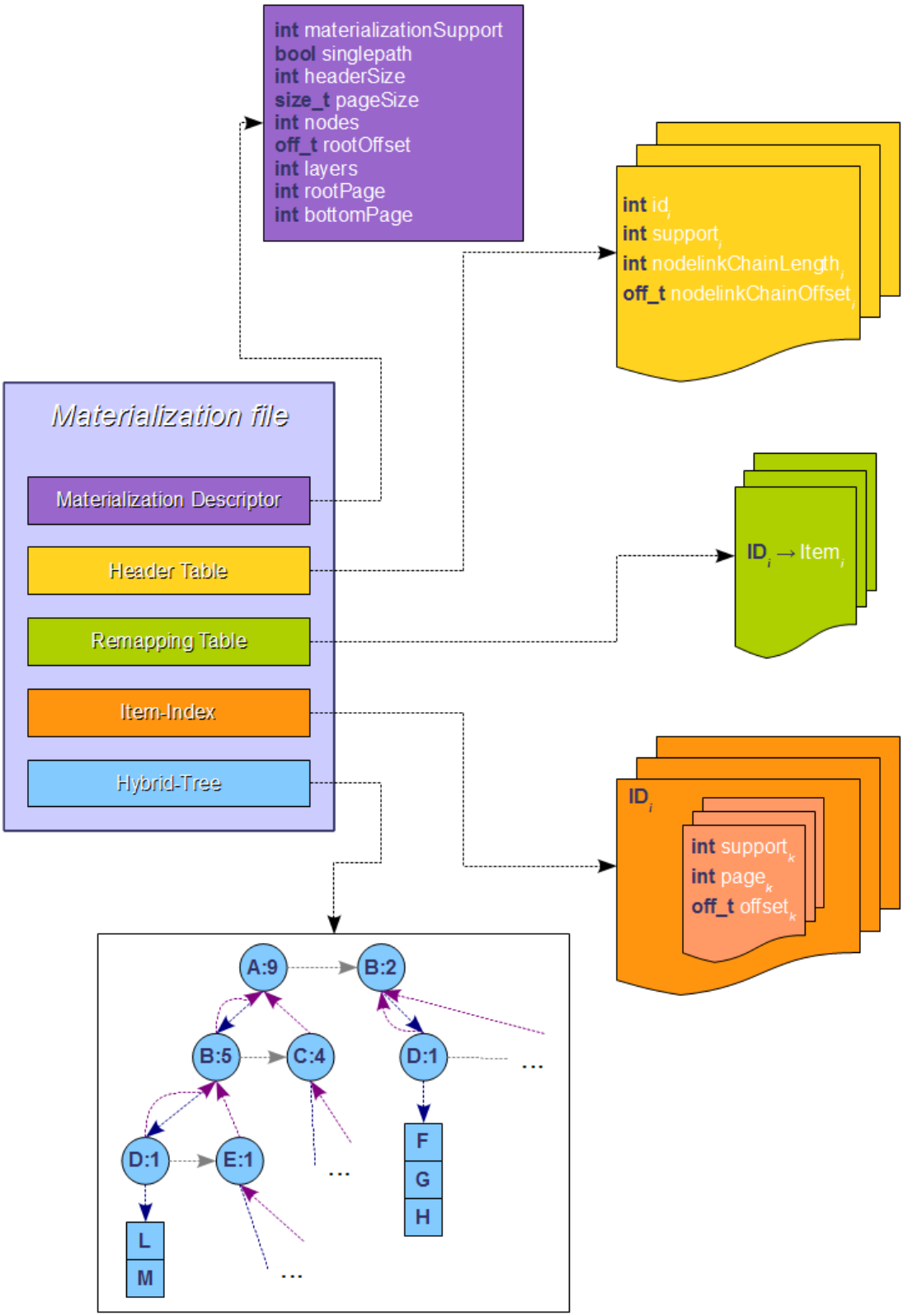

Figure 5. Materialization format

are just as common; they involve navigating all paths including a given item in a bottom-up way, to retrieve items along the way to the root, and, optionally, in a top-down way.

Based on these observations, nodes along each path are written on disk in adjacent physical locations by means of a depth-first visit, so as to maximize their contiguity and speed up vertical navigations.

CHAPTER 5

DATA ACCESS FRAMEWORK

The process of creating the Hybrid-Tree on disk and that of extracting useful information from it by means of mining algorithms are very I/O-intensive tasks. Due to the large amounts of data to be transferred from secondary storage into main memory and viceversa, optimizing the data access framework is vital to minimize the impact of I/O operations on the overall execution times.

To this aim, possible data access strategies have been analysed in Section 5.1 to identify their pro's and con's. The I/O framework implemented to access both the G-tree and the Hybrid-Tree is described in Section 5.2.

5.1 Possible I/O approaches

As outlined in (**?**), the low level data access can be carried out resorting to three main strategies:

- the standard I/O library (`fopen`, `fread`, `fwrite`, `fseek`);
- the UNIX I/O system calls (`open`, `read`, `write`, `lseek`);
- memory-mapped I/O.

5.1.1 The standard I/O library

The standard I/O library is specified by the ISO C standard because it has been implemented on a number of operating systems other than the UNIX System. For this reason, it provides the best portability among different platforms.

The standard I/O library handles issues such as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about such details. This makes the library easy to use but, on the other hand, introduces another set of problems. In particular, the different ways buffering is handled, depending on the type of stream, can generate confusion.

The major drawback of this approach, however, is the performance degradation it can introduce. In order to perform I/O operations, standard I/O library functions use an internal buffer whose size is by default very small. Because these functions rely on the OS's system calls to carry out the actual reading or writing task, this results in a large number of system call invocations, with significant additional overhead. Data is in some cases copied twice, from the kernel buffer to the library function buffer, and from the latter to the user buffer. Moreover, these functions conflict with the Virtual File System (VFS) buffers slowing down reading performance when disk pages are already virtualized by the operating system.

As a consequence, the C library functions are not a good choice to perform I/O of large amounts of data.

5.1.2 UNIX system calls

Compared to the previous approach, the use of the UNIX system calls can considerably reduce I/O times when large files are involved, due to the presence of a single, kernel-space buffering mechanism which enables reading one or more disk blocks at once.

The family of system calls (e.g., `read`) only uses the kernel VFS, which causes a low performance during the first reading, but speeds up all further read operations. As far as the writing phase is concerned, the `write` primitive is quite poor, having no support for delayed/asynchronous output.

5.1.3 Memory-mapped I/O

The memory mapping mechanism represents in many cases the most flexible choice for both reading and writing. A memory-mapped file is a segment of virtual memory which has been assigned a direct byte-for-byte correspondance with some portion of a file or file-like resource. This resource is typically a file that is physically present on disk, but can also be a device, a shared memory object or another resource that the operating system can reference through a file descriptor (in a UNIX environment) or a handle (in a Windows system). Most modern operating systems or runtime environments support some form of memory-mapped file access.

Memory-mapped I/O provides a versatile access to data. Programs exploiting memory-mapped file access can maintain dynamic data structures conveniently stored in permanent files and do not need to concern themselves with the movement of data between the file and the memory, which is handled by the operating system. There is thus no need to manage buffers,

and efficient in-memory algorithms can process file data, even though the file may be much larger than available physical memory, just as if they were operating in main memory.

In addition to the flexibility, using memory-mapped files provides increased I/O performance than direct read and write operations for the following reasons:

- system calls entail a large overhead and are orders of magnitude slower than a simple change of program's local memory;
- in most operating systems, the memory mapped region is actually the kernel's file cache, meaning that no copies are created in user space. Using system calls would inevitably involve the time-consuming operation of memory copying;
- since the memory-mapped file is handled internally in pages, sequential readings on a file require disk access only when a new page boundary is crossed, and writing large portions of the file on disk is performed in a single operation;
- applications can access and update data directly and in-place, as opposed to seeking from the start of the file or rewriting the entire edited contents to a temporary location.

A further possible benefit of memory-mapped files is the "lazy loading" technique, which saves significant amounts of RAM even for large-sized files. As a matter of fact, trying to load the contents of a file in whole can cause severe thrashing when the file is considerably larger than the amount of memory available: the operating system starts reading from disk to memory, but it will eventually end up simultaneously swapping from memory back to disk. Thanks to the

lazy loading, not only may memory-mapping completely bypass the page file, but the system will selectively load only those smaller page-sized sections of the file which are being edited.

These operations are handled by the virtual memory manager, which is the same subsystem responsible for dealing with the page file. Memory-mapped files are loaded into memory one entire page at a time, the page size being selected by the operating system for maximum performance. Since page file management is one of the core elements of a virtual memory system, loading page-sized sections of a file into physical memory is typically a highly optimized system function.

As a result, memory-mapped I/O often performs better than the other techniques. This approach, however, has its cost in page faults, which occur whenever a piece of data is required belonging to a block that has not yet been fetched from disk. Depending on the number of page faults, memory-mapped I/O can actually become substantially slower than standard file I/O. In order to reduce the chances of such an event, data should be stored in memory-mapped files trying to cluster blocks that will be read at the same time. In this respect, the depth-first serialization of the Hybrid-Tree is highly beneficial.

5.2 The BufferCache framework

The memory-mapped I/O approach has been selected over the C library and the UNIX system calls as the preferred data access method for this framework, which has its core in the BufferCache object.

The BufferCache is an object-oriented structure that carries out a role similar to the one of a DBMS buffer cache. It is exploited to access the nodes of the G-tree during the materialization

phase and those of the Hybrid-Tree during the mining process. These nodes are grouped in pages, each one called a MappedRegion. The BufferCache allows keeping a certain number of regions in memory, while directly managing the swap-in/swap-out of the other ones, according to frequency and chronology of accesses.

5.2.1 Memory-mapped areas

Each MappedRegion represents a mapped area of the materialization file. The MappedRegion is an object including the following pieces of information:

- the starting offset within the mapped file;
- a pointer to the mapped area;
- a collection of helper methods to manage the wrapped data.

Because the BufferCache is very likely to deal with files whose size exceeds 2 GB, it is necessary that the source code be compiled with the large file support. This way, the offset pointer is specified on 64 bits and is thus able to address files larger than 2 GB, even on 32-bit architectures.

In the initialization phase, a region inside the file is memory-mapped by invoking the `mmap` system call. The latter returns a pointer inside the process addressing space; this pointer will be used to access data within that disk region.

Special composite pointers are exploited to address the G-tree and Hybrid-Tree nodes, made up by two different references:

- a region number (16 bits);

- a relative pointer (32 bits), which corresponds to the offset inside the region.

Whenever a pointer to a node of the tree is needed, the BufferCache executes the following procedure:

1. it checks whether the addressed page is currently in memory;
2. if not, it loads it from disk, possibly swapping out the least used region;
3. it computes the real pointer by summing up the base address of the region and the relative pointer.

5.2.2 Swapping controller

The number of memory-resident regions is upper-bounded by a user-specified parameter (and by the available physical memory, of course!), while the decision to keep in or swap some of them out of RAM is taken by the BufferCache controller. This choice depends on two parameters:

- frequency of access to each region;
- chronology of access.

As a consequence, the more frequently a region is accessed, the higher the chances to keep (and hence to find) it in main memory.

To avoid multiple swaps of newly accessed areas, which are likely to be accessed a lot of times in the near future, the last swapped-in region is never discarded to make room for a new one, even if its access count is low.

The combination of these two parameters ensures a fair behavior of the BufferCache in the majority of cases.

5.3 Data access primitives

The LSD framework can support in principle any itemset extraction algorithm. Depending on the the enforced support threshold and the selected algorithm, a different portion of the materialized structure should be accessed. To support the tightly-coupled integration of the LCM v.2 mining algorithm, as described in Section 6.3, frequent-item projections must be extracted from the Hybrid-Tree. To this aim, two data access primitives have been developed:

- the Item-Index decompression primitive;
- the GetDenotation primitive.

5.3.1 Item-Index decompression primitive

This data access method can be exploited to retrieve data from the Item-Index and convert it into a usable form. Because of the compression technique adopted to compact the Item-Index chains, it is not possible to load this structure from disk into main memory and directly exploit it. The Item-Index decompression primitive takes care of the decompression phase in a seamless way and returns the node-link chain for a given item in an array form.

The primitive is invoked along with a parameter representing the item whose node-link chain shall be extracted. The header table of the materialization is accessed and the offset of the corresponding chain is read.

After that, the full compressed chain is loaded from disk. The decompression algorithm is subsequently applied and the uncompressed chain, containing standard composite pointers, is recreated in main memory. Because node-link chains are usually traversed one at a time and then discarded, they can be kept in memory in an uncompressed form, so that the decoding step is performed only once and the chain can then be navigated in any order.

5.3.2 GetDenotation primitive

The frequent-item projection of the dataset with respect to an arbitrary item α includes the transactions where α occurs, intersected with the items having higher support than α or equal support but preceding α in lexicographical order. Items along Hybrid-Tree paths are sorted by descending support and lexicographical order. As a consequence, the frequent-item projection is represented by the Hybrid-Tree prefix paths of item α (i.e., the subpaths from the roots to the nodes labelled with α).

The GetDenotation data access method reads the frequent-item projected database from the Hybrid-Tree. First, the Hybrid-Tree nodes including item α must be identified. To this aim, the Item-Index and the corresponding access primitive are exploited. Then, for each node, its prefix path is traversed by means of a bottom-up visit, following node parent pointers, until a tree root is reached. In the case where the node referenced by the node-link chain is a sentinel, the lower layer of the Hybrid-Tree shall also be accessed. All items comprised between the sentinel node and the node containing α are read by means of a single operation.

Once read, prefix paths are stored in an in-memory representation. In each prefix path, node supports are normalized to the node support of item α in the subpath, because only transactions including α must be taken into account.

The GetDenotation primitive has been optimized to enhance performance when many item-based projections must be sequentially extracted from the Hybrid-Tree. The node-link chain navigation is in fact carried out by inverting the direction at every iteration. Based on the fact that node-link chains are sorted by increasing tree page number, this technique tries to maximize the temporal locality of data accesses between contiguous iterations. Tree pages that were used last in the previous iteration have higher chances of still being in the BufferCache than pages that were used at the start of the iteration. As a consequence, starting the traversal from the tree portion that is still in memory avoids some disk reads and increases data re-use.

The GetDenotation primitive is illustrated in Algorithm 2.

Algorithm 2 Outline of the GetDenotation primitive

```
procedure GetDenotation (i: item) is
begin
  -- projection database
  P := ∅;

  chain = GetNodelinkChain (i);

  for j in 1..size(chain) do
    t := ∅; -- current transaction
    parent = chain[j].getParent (HybridTree);
    if parent.childrenChainLen > 0 then
    -- node is a sentinel
      len = parent.childrenChainLen;
      t := t ∪ ReadChildren (len);
      supp := 1;
    else
      supp := chain[j].support;
    end if;
    while parent not null do
      t := t ∪ parent.id;
      parent := parent.getParent (HybridTree);
    end while;
    t.multiplicity := supp;
    P := P ∪ t;
  end for;

  return P;
end GetDenotation;
```

CHAPTER 6

MINING ALGORITHM INTEGRATION

Frequent itemset mining can be carried out by exploiting existing mining algorithms, properly fed with the transactional data extracted from the Hybrid-Tree. Thanks to the omni-directional traversal provided by this structure, it would be possible, in principle, to support any mining algorithm.

Two main integration approaches can be followed.

The first approach is the least invasive for the mining algorithm. It consists in exploiting the disk-based tree as a pre-processing module: transactional data can be extracted from it for any mining threshold, providing a compact and pruned representation of the original dataset. The in-memory mining algorithm does not require any modification, but it is now able to exactly allocate its performing memory-resident data structures, thus reducing its main memory requirements. The scanning and pruning phases are, in fact, the memory hungriest ones and hence one of the major bottlenecks of traditional mining algorithms. Performing this task by means of the Hybrid-Tree makes it possible to mine larger datasets and for lower support thresholds.

The second approach requires a modification of the mining algorithm and can affect the way the transactional data is retrieved from the tree and passed to the mining algorithm. In the previous case the entire dataset was extracted in a pruned form and sent to the mining algorithm in whole. In contrast, the alternative approach lies in selectively loading specific

portions of the dataset only when needed. These portions can be chosen according to different criteria, depending on the type of traversal required by the mining algorithm we are trying to integrate. For this reason, a deeper knowledge of how the algorithm itself operates is required.

Both approaches have their advantages and drawbacks. The former approach is the simplest to implement, but also the first to fail when mining of huge datasets is addressed. The reason is that even the pruned dataset extracted from the tree may at some stage not fit in main memory, thus making the memory-based mining process impossible. The latter approach is likely to overcome this limitation, at the cost of an increased integration complexity. The LCM v.2 algorithm has been integrated following this strategy.

6.1 LCM: an overview

As described in (3), LCM v.2 is an efficient, linear-time itemset mining algorithm that currently outperforms all other state-of-the-art approaches. Its driving forces are the exclusive use of array-based structures and a suite of clever techniques which reduce the computation time efficiently.

The basic idea of the algorithm is a depth-first search. Let us consider a dataset $T = \{t_1, t_2, \ldots, t_m\}$, t_i being the i-th transaction in T. We define the *denotation* of an itemset P, and note it as $T(P)$, to be the subset of transactions in T containing itemset P, namely $T(P) = \{t_j \in T | P \subseteq t_j\}$. Further, we denote the largest item (i.e., the item with the highest frequency) of an itemset P by $tail(P)$.

LCM first computes the frequency of each itemset composed of one item. If an itemset $I = \{i\}$ is frequent, then it enumerates frequent itemsets obtained by adding one item to $\{i\}$.

Algorithm 3 Trivial implementation of DFS frequent itemset mining

```
procedure FrequentItemsetMine (D: database; I: itemset) is
begin
  OutputItemset (I);
  for j in tail(I)+1..N do
    if I∪{j} is frequent then
      FrequentItemsetMine (D, I∪{j});
    end if;
  end for;
end FrequentItemsetMine;

procedure Mine is
begin
  FrequentItemsetMine (D, ∅);
end Mine;
```

In this way, LCM can recursively enumerate all frequent itemsets. In order to split the search space in non-overlapping partitions, and hence avoid computing duplicate itemsets, only items $j > tail(I)$ (i.e., more frequent than i, in this case) are considered.

A straightforward implementation of this approach in given in Algorithm 3. However, such an implementation is very slow, because computing the frequency of $I \cup \{j\}$ requires an entire pass over the input dataset. In order to speed up this computation, LCM resorts to the *conditional database* and *occurrence deliver* techniques.

6.1.1 Conditional database

The *conditional database* is the first enhancement introduced by LCM which signficantly contributes to speeding up the mining process. The conditional database of itemset I corresponds to its denotation $T(I)$, after additionally removing all unnecessary items from each

transaction and merging all identical transactions into one. An item j is considered *unnecessary* provided that it satisfies one of the following conditions:

1. $|T(I \cup \{j\})| < suppThreshold$ (it is included in less transactions than the user-specified threshold);
2. $T(I \cup \{j\}) = T(I)$ (it is included in all transactions containing in I);
3. $j < tail(I)$.

Once the conditional database has been constructed, the frequency of $I \cup \{k\}$, $k > tail(I)$, is the same in both the original and the conditional database. As a consequence, LCM uses the latter, rather than the former, in the recursive call with respect to I.

6.1.2 Occurrence deliver

The *occurrence deliver* technique is an efficient method to compute the denotation of an itemset.

Occurrence deliver is able to find the denotations of all itemsets $P \cup i$ at once, where P is initially the empty set and i is any item such that $i > tail(P)$. To this aim, an array-based structure containing a bucket for each possible item is exploited; buckets are initialised to the empty set. The conditional database of the dataset with respect to P is scanned once, transaction by transaction and, for each item $i > tail(P)$, the ID of the corresponding transaction is inserted into the item bucket. At the end of this process, each bucket i contains the denotation of itemset $P \cup \{i\}$.

In order to recursively compute the denotation of the itemsets obtained by adding new items, it is sufficient to replace P with $S = P \cup \{i\}$, calculate the conditional database and repeat the above operation.

This procedure is linear in the sum of the sizes of the transactions included in the conditional database and it is one of the key routines of the LCM algorithm.

6.2 LCM technicalities

This section aims at providing a more in-depth view on the LCM implementation, useful to understand how this algorithm can be tightly integrated in the Hybrid-Tree framework.

6.2.1 Pre-processing

Most frequent itemset mining algorithms need to perform some pre-processing steps on the input dataset, prior to the actual mining phase, and LCM is no exception.

The original, plain-text dataset is initially scanned once to count the number of transactions and the frequency of each distinct item it includes. Infrequent items are pruned with respect to the user-specified support threshold, `LCM_th`; items are sorted by decreasing global support and assigned a unique progressive ID, thus defining a remapping.

The dataset is then read a second time and the frequent items of each transaction (if any) are stored in main-memory in array form.

Finally, identical transactions are merged into a single one, whose multiplicity is set accordingly. To this aim, an efficient technique is employed, linear in the size of the dataset and based on a radix-sort strategy.

After that, the pre-processing phase is complete and a number of auxiliary, memory-resident data structures are allocated, including the `LCM_Occ` structure, necessary for the *occurrence deliver* routine.

6.2.2 Mining procedure

The actual mining task has its core in the `LCMfreq` procedure. Prior to its invocation, the occurrence deliver routine needs to be called once, so as to populate the `LCM_Occ` structure.

The `LCMfreq` procedure consists of a loop in which items are considered one by one. For each of them, the `LCMfreq_iter` recursive procedure is called. It operates as follows.

Given item e, the set of items co-occurring with e is computed, by scanning transactions in $T(\{e\}) \equiv$ `LCM_Occ[e]`, and stored in a queue, called the *jump queue*. As previously described, only items with a higher frequency than $tail(e)$ are considered. Due to the remapping defined in the pre-processing step, this simply means that only items in $\{0, \ldots, e-1\}$ are taken into account in the iteration with respect to e.

The conditional database is then computed. The *jump queue* is pruned of all items that co-occur with e less than `LCM_th` times and thus cannot originate any frequent itemset together with e. In addition, items that appear in all transactions in $T(\{e\})$ are removed from the *jump queue* and inserted into a special queue, the *add queue*. This is to avoid useless computations, since these items are already sure to participate in the frequent itemsets generated from item e. They are added back at the end of the recursion (hence the name "add queue"), when discovered frequent itemsets are output.

If at this stage the jump queue is empty, meaning that no items other than unnecessary ones are present in the conditional database, the iteration with respect to item e terminates and the algorithm moves on to the next item. Otherwise, the conditional database is projected into a temporary memory structure and `LCMfreq_iter` is called on this database for each of the items inside the jump queue.

When only few items are co-frequent with e, the benefit provided by this procedure can be overshadowed by the cost of the recursion. For this reason, recursive calls are not made if the jump queue contains less than three items; special functions are called instead to handle the mining process in an iterative fashion. Likewise, identical transactions are not collapsed into one upon creating the conditional database if the denotation is small. These clever optimizations can significantly improve the performance of the extraction task when the dataset is sparse and most of the mining steps are dealing with few items.

6.3 LCM integration

As reported by the author of the algorithm, LCM requires an amount of physical memory which roughly corresponds to as many integers as three times the database size (given by the sum of all transaction lenghts). Because LCM allocates all the memory it will ever need during the initialization, it will either fail in this phase or succeed in the mining task. This stable memory usage is an advantage compared to other algorithms, such as FP-growth, which can fail at any time throughout the mining process because they cannot allocate any more memory, thus wasting CPU (and user!) time.

Nonetheless, when the input database comprises tens of millions of transactions, memory usage becomes an issue. The LCM v.2 algorithm has been included in the framework following a tight integration approach, in an attempt to reduce its memory requirements.

6.3.1 A memory-sparing approach

The implemented integration strategy is grounded on the observation that the LCM algorithm intrinsically operates with a partitioning approach. As previously outlined, the algorithm's main loop selects one item at a time and calls the recursive mining procedure, which performs its computations on the conditional database with respect to that item. As a consequence, when iterating on item i, no other information than its denotation is required, and the rest of the dataset needs not be in main memory. This feature of the algorithm has been exploited to reduce its memory footprint.

Rather than the entire transactional dataset, the denotation of a single item at a time is extracted from the Hybrid-Tree. This is possible thanks to the item-based traversal enabled by the Item-Index. The recursive mining procedure is subsequently called on this item-based projection. Upon returning from the recursion, the item-based projection is removed from main memory and the next one is fetched from the disk-based tree. This approach ensures that only the data actually required by the mining process at any time are present in main memory. Since item-based projections are in most cases orders of magnitude smaller than the full dataset, a significant memory saving can be achieved, thus enabling a broader-scale exploitation of the LCM algorithm.

6.3.2 Implementation

To the aim of integrating it into the LSD framework, the LCM source code has been decomposed and some portions have been moved inside the framework. The LCM code thus runs as part of the framework process, and not as a separate one. This solution has been chosen for maximum performance, because it makes it possible to directly handle the memory-resident structures from within the framework. The alternative to a single process would have required some inter-process communication strategy, such as pipes or shared memory. However, these solutions are costly: the former introduce the overhead of data copying between processes; the latter necessitates additional synchronization primitives, such as semaphores, thus further complicating the integration. With the adopted strategy, the algorithm simply runs on its usual data structures, which are populated by the framework as appropriate.

As with the original in-memory version, two main steps can be identified: a pre-processing phase and the actual mining phase.

Pre-processing

In the context of the Hybrid-Tree framework, the original pre-processing step performed by LCM is to a large extent unnecessary, because the transactional data stored in the tree is already organised following the same criteria. Hence, some of LCM's data structures are directly allocated and filled by the framework. These structures include:

- the item remapping table, `LCM_perm`;
- the transaction array, `LCM_trsact`;
- the `LCM_occ` array structure, used for the occurrence deliver.

Prior to all operations, the header table of the tree is pruned of infrequent items.

The item remapping table is then simply copied from the one included in the materialization file.

The last two structures, on the other hand, require a more careful analysis in order to allocate a sufficient amount of memory.

Because the transaction array must be able to (separately) accommodate the denotations of all items, its size must equal that of the largest denotation; this is to avoid the costly operation of reallocating memory throughout the mining process. This information can be obtained from the header table of the Hybrid-Tree.

More precisely, the header table contains the entry point to and the length of the Item-Index chains. Every node referenced by the Item-Index chains defines a distinct path in the tree, from the root to the node itself. As a consequence, the item denotation will include at least one transaction beginning with each of such prefix paths. If the node at stake is not a leaf, multiple transactions will then share the same prefix path. It thus appears that the length of a given Item-Index chain is a lower bound for the size of the corresponding denotation.

The conditional database, computed from the denotation, will include at least as many distinct transactions. In addition, it must be recalled that, in order to obtain the conditional database, all items with a lower global frequency than the considered one need to be removed from the denotation. Due to the way transactions are sorted in the tree paths, this translates to discarding all nodes located below the one referenced by Item-Index chain. Consequently, the above mentioned prefix-paths are exactly the transactions included in the conditional database;

the length of the chain thus represents the exact size of the conditional database for a given item.

It is therefore sufficient to look for the maximum Item-Index chain length in the header table and tailor the `LCM_trsact` structure accordingly.

After that, the `LCM_occ` structure is allocated. This structure consists of a number of arrays, one for every possible (globally frequent) item. Each array cell is used to store the ID of a transaction containing a given item and it is populated prior to each iteration. In principle, it would be best to allocate for each item as many array cells as the maximum number of transactions in which it can appear, across all denotations. However, this number is not known in advance and it cannot be computed unless all item-based projections of the dataset are also computed and read once, prior to the allocation. To avoid this useless computational effort, a loose bound is exploited, corresponding to the previous "max-length" bound. By doing so, one can be sure that all size constraints on `LCM_occ` are satisfied in all denotations. The drawback is that some memory is thus wasted. However, this waste is always upper-bounded by the size of the largest item-based projection, which is a great deal smaller than the overall number of transactions. In most practical cases, the amount of wasted space can be considered negligible.

The rest of the data structures comprises a number of auxiliary buffers. Since no particular value or size estimate is needed to initialize them, the original `LCM_init` routine is let take care of their allocation.

Mining

Once all data structures have been conveniently created, the actual mining process can take place. To this aim, one denotation at a time is extracted from the tree resorting to the GetDenotation primitive described in Section 5.3. As formerly explained, items smaller than the current one (i.e., located farther from the root) are not read and hence no bottom-down traversal is necessary. Not only does this save on I/O costs, but it eases the subsequent conditional database reduction performed by LCM because it removes unnecessary items beforehand and contextually collapses identical transactions into one thanks to the common prefix they share.

The `LCM_occ` structure is then populated by filling the array of the current item, i, with the IDs of the transactions in which it appears. Since the algorithm is dealing with item-based projections, this boils down to writing numbers in $\{0, \ldots, |T(i)| - 1\}$. The `LCMfreq_iter` procedure is then called with respect to item i and the mining task is performed on the current denotation.

A sketch of the integrated mining procedure is provided by Algorithm 4.

Algorithm 4 Outline of LCM disk-based mining

```
procedure LCMDiskBasedMine (σ: support) is
begin
  AllocateDataStructures ();
  I := PruneItems (σ);
  for item i in I do
    D := GetDenotation (i);
    for j in 1..size(D) do
      Occurrence[j] := j;
    end for;
    LCMMineInMem (D, Occurrence);
    Delete (D);
  end for;
end LCMDiskBasedMine;
```

CHAPTER 7

MATERIALIZATION PERFORMANCE

This chapter is meant as an evaluation of the contributions provided by the first part of this master thesis work.

The focus in this part was on the development of a scalable sorting strategy to perform the dataset pre-processing step and a compression technique to reduce the size of the Item-Index.

The following sections will analyze the features of the materialized structure comprising the Hybrid-Tree and the Item-Index for a number of synthetic datasets generated with different parameters. The leverage of the sorting operation on the overall creation time will also be discussed.

7.1 Materialization features

In order to validate the proposed approach, the performance of the Hybrid-Tree and Item-Index structures has been addressed by performing a large set of experiments. Different representative synthetic datasets have been chosen to this aim, whose characteristics in terms of transaction and item cardinality, length and correlation of frequent patterns and dataset size are shown in Table II.

All datasets have been created by means of the IBM synthetic dataset generator, by setting different parameters (i.e., T average transaction length, P average maximal pattern length, I number of different items, C correlation grade between patterns, and D number of transactions).

Experiments have been run on two different machines, depending on the size of the dataset. In particular, experiments on datasets including less than 45M transactions have been run on a commodity PC with the following characteristics (configuration I):

- Intel(R) Pentium(R) 4 3.20GHz processor
- 2.5 Gbyte main memory
- Linux kernel 2.6.20.

In contrast, experiments on datasets whose size exceeds 45M transactions have been performed on a more powerful machine (configuration II):

- Intel(R) Core(TM) 2 Quad 2.66 GHz processor
- 8 Gbyte main memory
- Linux kernel 2.6.28

7.1.1 Data compaction

In Table III the materialization characteristics of the considered datasets are given. In order to evaluate the degree of compaction provided by the hybrid structure, a compression factor (CF) has been defined as follows:

$$CF = \left(1 - \frac{size\,(\mathsf{Hybrid\text{-}Tree}) + size\,(\mathsf{Item\text{-}Index})}{size\,(Dataset)}\right)\%$$

It compares the overall size of the materialization with the size of the transactional dataset. The compression factor increases when the fraction of upper-layer nodes is high, because upper-

Dataset	Size (gb)	# Transactions	# Items
T20P20I100kC0.75D10M	1.40	10M	49,372
T22P22I50kC1D15M	2.27	15M	32,515
T22P20I300kC0.75D20M	3.32	20M	73,656
T24P24I300kC1D25M	4.74	25M	75,196
T22P22I50kC1D30M	4.54	30M	32,520
T22P20I250kC0.5D45M	7.37	45M	80,873
T20P18I150kC0.75D60M	8.47	60M	55,900
T20P18I150kC0.75D100M	14.12	100M	55,900
T20P18I150kC0.75D500M	70.60	500M	55,900
T20P18I150kC0.75D1000M	141.19	1000M	55,900

TABLE II

DATASET CHARACTERISTICS

Dataset	UL Nodes (%)	LL Nodes (%)	Tree size (gb)	Index size (gb)	Total size (gb)	CF (%)
T20P20I100kC0.75D10M	12.54	87.46	0.63	0.72	1.36	3.24
T22P22I50kC1D15M	14.34	85.66	1.08	1.16	2.24	1.55
T22P20I300kC0.75D20M	17.40	82.60	1.59	1.55	3.14	5.17
T24P24I300kC1D25M	18.09	81.91	1.99	1.91	3.89	17.93
T22P22I50kC1D30M	18.64	81.36	2.23	2.11	4.34	4.38
T22P20I250kC0.5D45M	21.64	78.36	3.66	3.18	6.85	7.13
T20P18I150kC0.75D60M	24.67	75.33	4.41	3.55	7.96	6.04
T20P18I150kC0.75D100M	27.74	72.26	7.02	5.27	12.30	12.91
T20P18I150kC0.75D500M	30.70	69.30	24.49	17.23	41.72	40.90
T20P18I150kC0.75D1000M	29.92	70.08	40.13	28.72	68.85	51.23

TABLE III

MATERIALIZATION CHARACTERISTICS

layer nodes can compactly encode a large number of transactions that share many items. The second and third columns illustrate how data has been split up between the two Hybrid-Tree layers, and hence provides information on the data distributions of the diverse datasets.

As it appears from the first lines of the table, small- and medium-sized datasets (10M, 15M, 20M, and 30M transactions) have only a small percentage of nodes residing in the upper layer, the majority of the tree being composed of lower-layer nodes. As a consequence, the compression factor is quite low for these datasets and the Hybrid-Tree and Item-Index structures provide a limited degree of compaction.

One noteworthy exception is represented by the T24P24I300kC1D25M dataset, whose compression factor (nearly 18%) is significantly better in comparison to the other datasets. The reason is that frequent patterns in this dataset are very long, which translates to a higher fraction of each transaction containing frequent items and, hence, to a larger part of the dataset being represented by means of upper-layer nodes.

It can thus be inferred that the Hybrid-Tree structure is most suitable for dense datasets, although it can still provide some degree of compression with sparser datasets thanks to the smaller-sized lower-layer node structures.

The percentage of upper-layer nodes tends to increase as the size of the dataset grows larger, as it is the case with the last datasets included in Table Table III. For such datasets, the compression factor reaches very high values, thus suggesting that the Hybrid-Tree representation is optimal for very large datasets.

The Item-Index size is reported in the fifth column of the table. In spite of the adopted compression technique, the final size of the Item-Index exceeds that of the tree for the first two datasets. However, as the cardinality of datasets increases, the Item-Index becomes smaller in size than the Hybrid-Tree. To evaluate the relative incidence of the Item-Index on the final materialization size and compare its size to that of the Hybrid-Tree, the following quantities have been considered:

$$ITR = \frac{size\,(\text{Item-Index})}{size\,(\text{Hybrid-Tree})}\% \text{ (Item to Tree Ratio)}$$

$$IMR = \frac{size\,(\text{Item-Index})}{size\,(\text{Hybrid-Tree}) + size\,(\text{Item-Index})}\% \text{ (Item to Materialization Ratio)}$$

The corresponding values for the considered datasets are provided in Table IV.

Starting from the 20M transaction dataset, the incidence of the Item-Index is subjected to a steady decrease. This phenomenon can be explained as follows. Both the Item-Index and the Hybrid-Tree sizes are proportional to the number of nodes in the tree. The Hybrid-Tree comprises two types of nodes – upper-layer nodes and lower-layer nodes –, which take up 28 bytes and 4 bytes respectively. For this reason, the Hybrid-Tree size is kept down to a limit even when the number of created nodes is very high. The Item-Index, in contrast, cannot provide such distribution-adaptive layering and all of its nodes have a constant size of 8 bytes. As a consequence, for sparser datasets, where lower-layer nodes predominate, the double-layering of the tree achieves better compaction than the Item-Index.

Dataset	ITR (%)	IMR (%)
T20P20I100kC0.75D10M	114.14	53.26
T22P22I50kC1D15M	107.51	51.78
T22P20I300kC0.75D20M	97.86	49.43
T24P24I300kC1D25M	95.91	48.94
T22P22I50kC1D30M	94.40	48.55
T22P20I250kC0.5D45M	87.02	46.51
T20P18I150kC0.75D60M	80.63	44.63
T20P18I150kC0.75D100M	75.06	42.87
T20P18I150kC0.75D500M	70.37	41.30
T20P18I150kC0.75D1000M	71.55	41.71

TABLE IV

INCIDENCE OF THE ITEM-INDEX

On the other hand, the Item-Index becomes significantly smaller than the Hybrid-Tree when larger datasets are concerned.

On the whole, the proposed materialized structure performs best with large datasets and, for a given dataset size, maximum compaction is achieved with the denser datasets.

7.2 Materialization time

The overall time needed to create a complete materialization starting from a raw dataset is the sum of five distinct contributions:

1. pre-processing step, which includes item support count and dataset remapping;
2. dataset sorting
3. construction of the G-tree;

4. optimization of data contiguity, creation of the Item-Index and lower layer;

5. serialization of the upper layer.

To the aim of evaluating the leverage of the sorting operation on the entire creation task, steps 4 and 5 will be considered as a whole and simply referred to as "materialization".

7.2.1 Sorting operation impact

The sorting algorithm of choice was initially the Linux `sort` utility, as discussed in Section 4.3.1. Its performance massively decreases as the dataset size grows. As shown by experiments, the time required to carry out the first two of the previously outlined steps (i.e., pre-processing and sorting) resorting to this algorithm accounts for about 55%-60% of the overall creation time.

The designed sorting algorithm has been integrated in the framework and tests have been run on different datasets. Figure 6, Figure 7 and Figure 8 highlight the time contribution of each step to the creation phase. Contributions relative to the last two datasets (500M and 1000M transaction) have been depicted separately being out of scale with respect to the smaller ones.

As it can be noticed, the developed algorithm much alleviates the sorting task. The impact of the the first two steps amounts to about 14%-18% on the smaller datasets and 25%-30% on the larger ones – the actual sorting step always being the least time-consuming one.

It is worth mentioning that the relative contribution of the sorting step gradually increases with the size of the dataset, ranging from 4% on the smallest dataset to nearly 10% on the largest one. This is actually due to the data distribution, rather than to some inefficiency

of the mining algorithm. As a matter of fact, the durations of the various creation steps are proportional to different quantities. In particular, the durations of pre-processing and sorting steps are directly proportional to the cardinality of the input dataset. The time needed for constructing the G-tree depends on both the input dataset and its data distribution: for denser datasets, less nodes are crated and hence less time is spent with disk write operations, thus speeding up the tree construction. Finally, all subsequent phases (tree serialization, Item-Index creation) depend solely on the number of nodes included in the G-tree.

As previously discussed, the Hybrid-Tree representation provides better compaction with larger datasets, meaning that less nodes are created for such datasets in comparison to the smaller ones. As a consequence, the impact of such phases tends to decrease on the larger datasets, while that of the pre-processing and sorting steps does not.

7.2.2 Algorithm comparative test

In order to further investigate the highlights of the developed sorting algorithm, some experiments have been performed by running the two algorithms on different datasets. The considered datasets differ from the ones that have been analysed up to now; in particular, because the sorting operation with the Linux `sort` utility requires much time, only smaller datasets have been employed for this experiment.

As reported in Table V, results show that the speed-up achieved by the developed sorting algorithm is always more than ten-fold. Although comparison has not been established on larger datasets, this speed-up factor is expected to increase significantly as datasets grow. The reason is that the Linux `sort` performance dramatically suffers from memory shortage and becomes

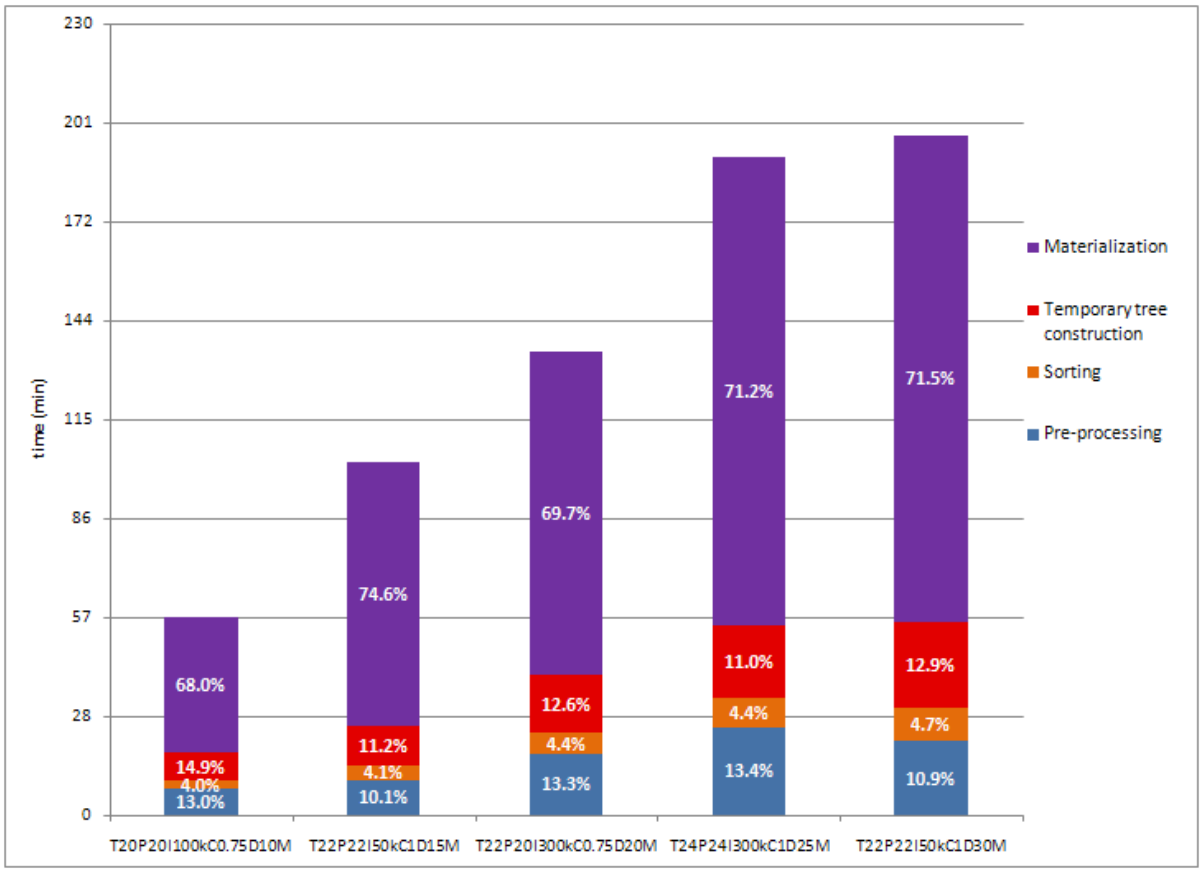

Figure 6. Materialization time for 10M, 15M, 20M, 25M and 300M transaction datasets

Dataset	Unix sort (s)	New algorithm (s)	Speed-up (x)
T20P20I100kC0.75D5M	923	39	23.67
T20P20I100kC0.75D10M	2134	140	15.24
T22P20I300kC0.75D20M	4116	356	11.56
T24P24I300kC1D25M	7066	511	13.83

TABLE V

SORTING TIME WITH DIFFERENT ALGORITHMS

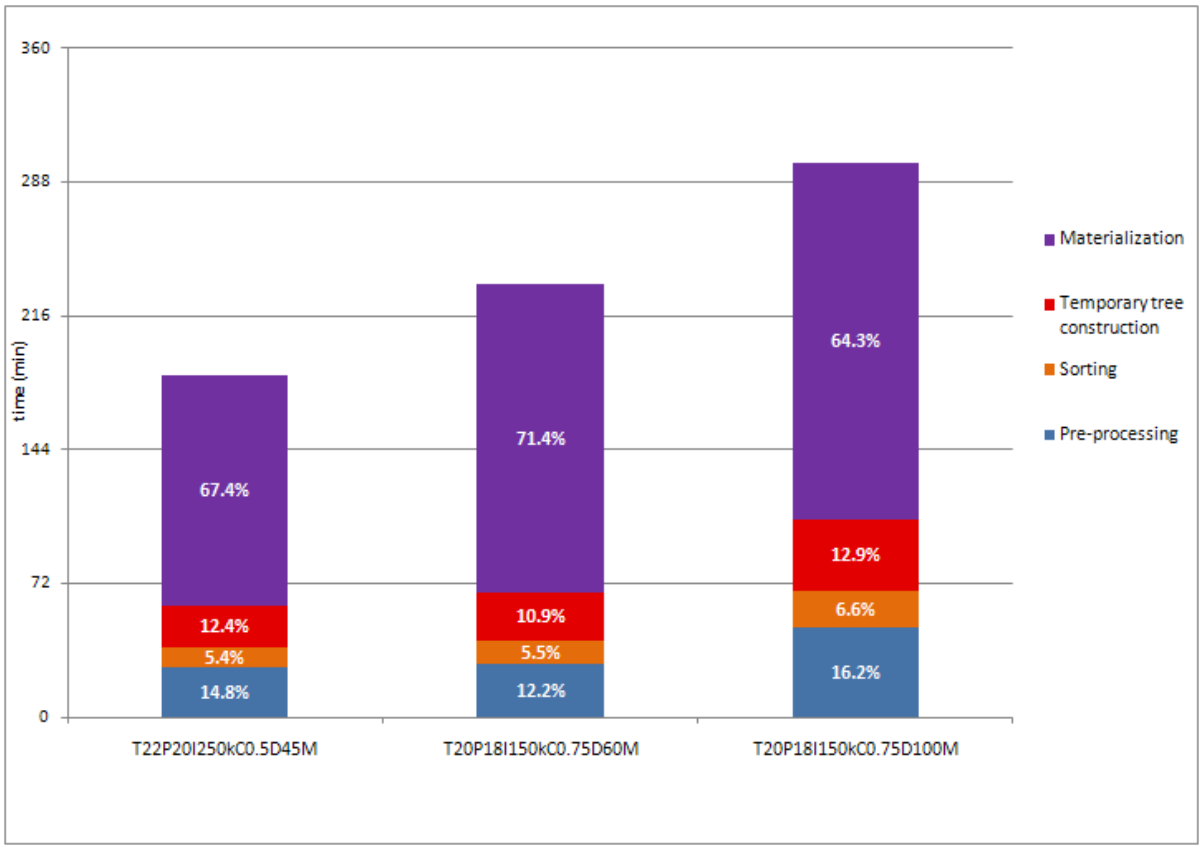

Figure 7. Materialization time for 45M, 60M and 100M transaction datasets

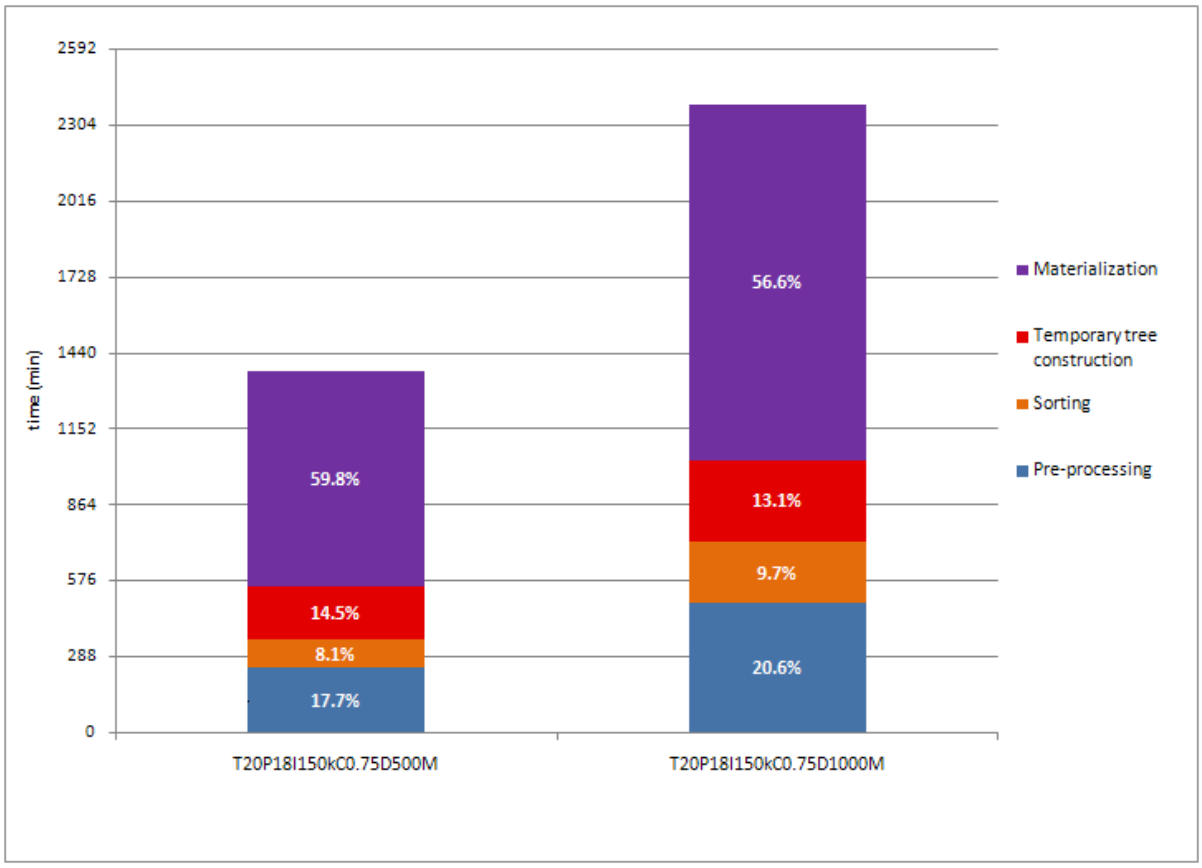

Figure 8. Materialization time for 500M and 1000M transaction datasets

extremely poor when much swapping occurs. This issue does not affect the developed sorting algorithm, which therefore exhibits much better performance.

7.2.3 Sorting algorithm scalability

As discussed in Section 4.3.2, the proposed sorting algorithm has a theoretical time complexity which is linear in the size of the dataset. In order to assess this property, a scalability test has been performed by comparing the sorting algorithm performance on a number of different-sized datasets.

Figure 9 and Figure 10 depict the results obtained with these experiments. As expected, the elapsed time grows linearly with the size of the considered dataset.

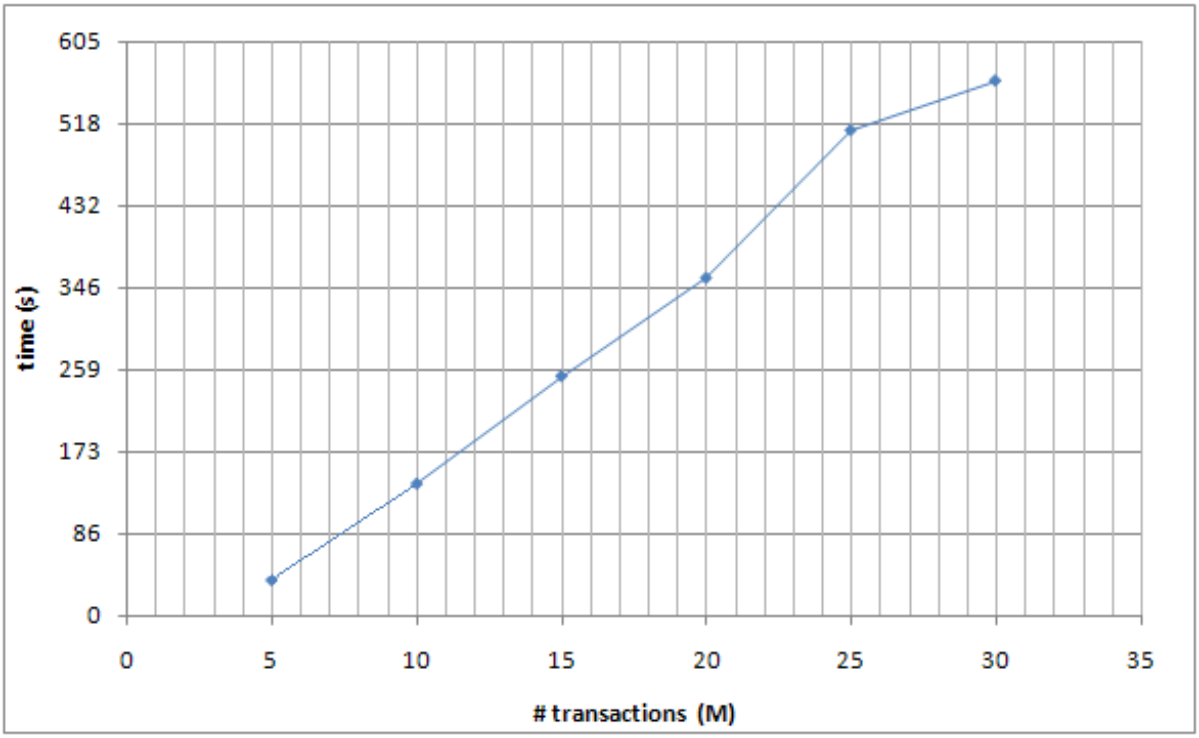

Figure 9. Sorting algorithm scalability on smaller datasets

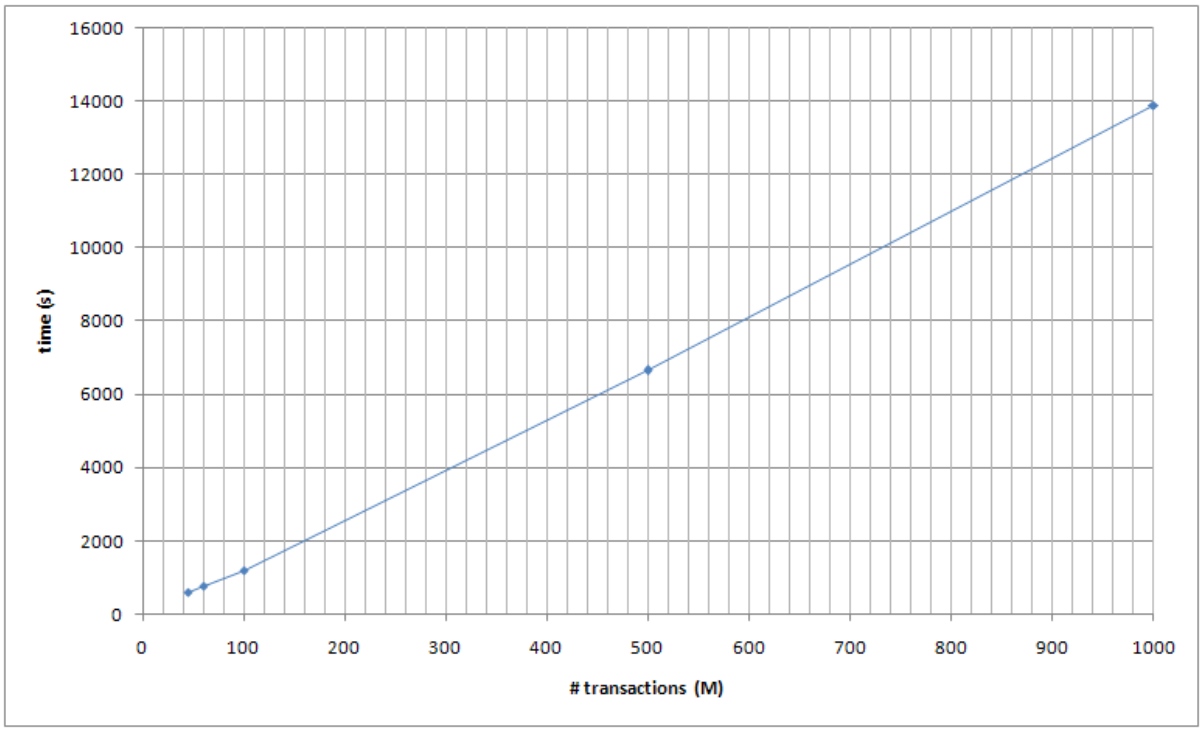

Figure 10. Sorting algorithm scalability on larger datasets

CHAPTER 8

MINING PERFORMANCE

This chapter aims at providing a detailed view on the performance of the mining task performed by means of the LCM v.2 algorithm integrated in our framework.

The first set of experiments is focused on the behavior of the disk-based integration of LCM v.2 when compared to the original memory-based counterpart. Such an approach is possible as long as the considered dataset can be accommodated in main memory for the mining process.

For larger datasets this approach is no longer feasible: the in-memory algorithm starts swapping to disk and, depending on the amount of available swap disk space, eventually fails in allocating the memory it needs. As a consequence, only the performance and scalability of the integrated version have been evaluated for these datasets.

Hereinafter, "LCM disk-based" will be used to designate the integrated version of LCM.

8.1 Small datasets

Three small datasets have been considered for this test case:

- T20P20I100kC0.75D10M
- T22P20I300kC0.75D20M
- T24P24I300kC1D25M

These datasets have in common the fact that they fit in main memory. Hence, this test case will illustrate the behavior of the two LCM versions when the in-memory version needs not swap to disk.

8.1.1 Results

T20P20I100kC0.75D10M

Results are depicted in Figure 11. For the considered support thresholds, the best performance is achieved by LCM disk-based. In particular, with with the higher support thresholds (0.3%-0.5%), the LCM disk-based significantly outperforms LCM in-memory. This is because the in-memory version has to read the entire dataset at least once and accomplish preliminary operations (item support counting, infrequent item pruning, transaction sorting). In contrast, the disk-based version needs no pre-processing and can thus directly start mining.

This gap between the memory-based and the disk based version mining times tends to decrease as the support threshold is lowered. For one intermediate threshold (0.25%), LCM in-memory performs slightly better than LCM disk-based.

Some jittering may be noticed in the LCM disk-based mining times. In particular, for two support threshold pairs (0.25%-0.20% and 0.15%-0.125%) mining time decreases as the threshold is lowered. This phenomenon can be explained by recalling that most operating systems provide a file cache mechanism. To reduce the costly operation of reading from disk, portions of frequently accessed files are cached in main memory, so as to speed-up future accesses to such files. It is reasonable to assume that the reduced mining time in the two aforementioned

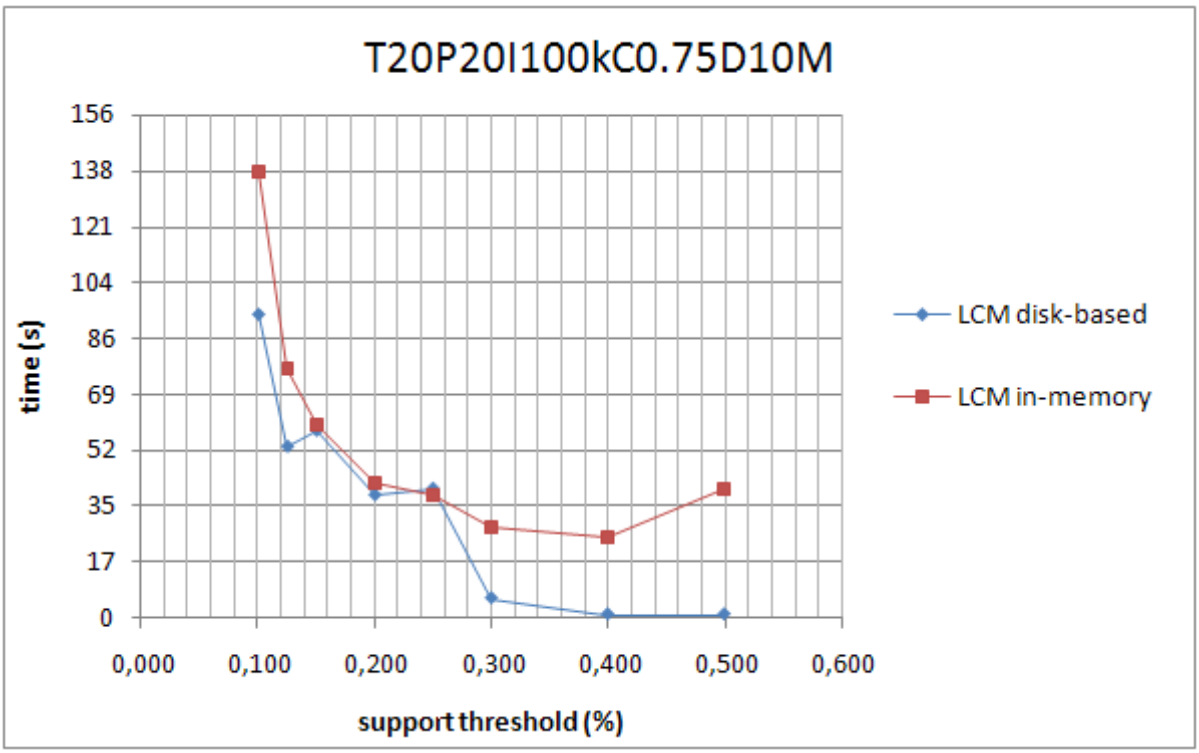

Figure 11. T20P20I100kC0.75D10M: LCM in-memory vs. disk-based

cases be due to tree pages (or parts of them) still being present in the operating system file cache, thus saving on I/O costs.

T22P20I300kC0.75D20M

Mining times for this dataset are plotted in Figure 12. Noticeably, LCM disk-based outperforms its in-memory counterpart by two orders of magnitude for the higher support thresholds. This advantage, which is quite constant until the 0.2% threshold, subsequently starts decreas-

ing as the support threshold is lowered. At a support threshold of 0.125% LCM disk-based eventually exhibits worse performance than the in-memory version.

This is due to the larger amount of data involved in the mining process and, hence, of disk reads that are needed by the former version. On the other hand, this phenomenon does not affect the latter version, because once the pre-processing step is complete, the full (pruned) dataset is in main memory and hence data accesses are much faster.

At the lowest support threshold the two versions are comparable again, probably because of both the operating system file cache and the smaller amount of processing required by the disk-based version.

T24P24I300kC1D25M

Results of this test are provided in Figure 13. This dataset has already been considered in Section 7.1.1, when discussing the performance of the materialized structure in terms of data compaction. As previously pointed out, the corresponding materialization presented a very good degree of compaction. This characteristic can be identified as the main reason for the good performance it also exhibits in the mining phase.

LCM disk-based outperforms LCM in-memory for all of the considered support thresholds. Thanks to the high compaction degree, only smaller amounts of data (in comparison with the plain-text dataset) need to be fetched from disk for the mining process. As a consequence, LCM disk-based still performs better than the in-memory version for extremely low support thresholds, such as 0.08%. Even though the number of frequent itemsets (greater than 10^9)

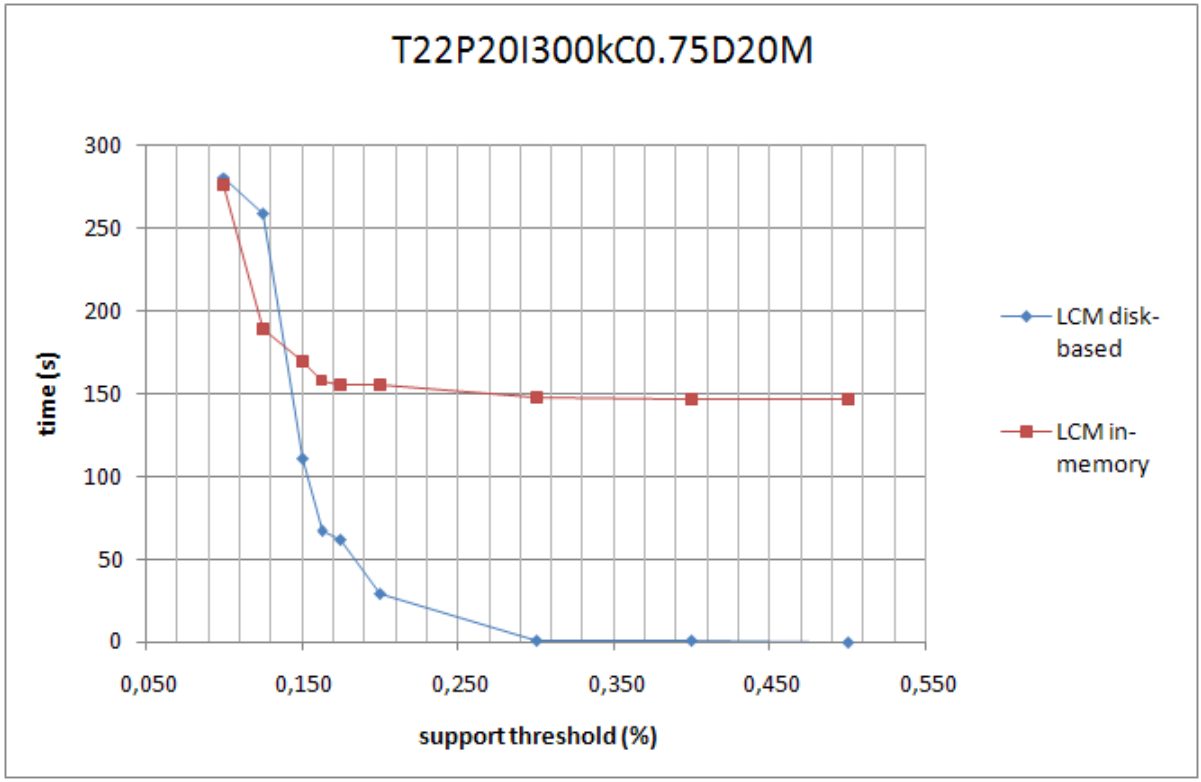

Figure 12. T22P20I300kC0.75D20M: LCM in-memory vs. disk-based

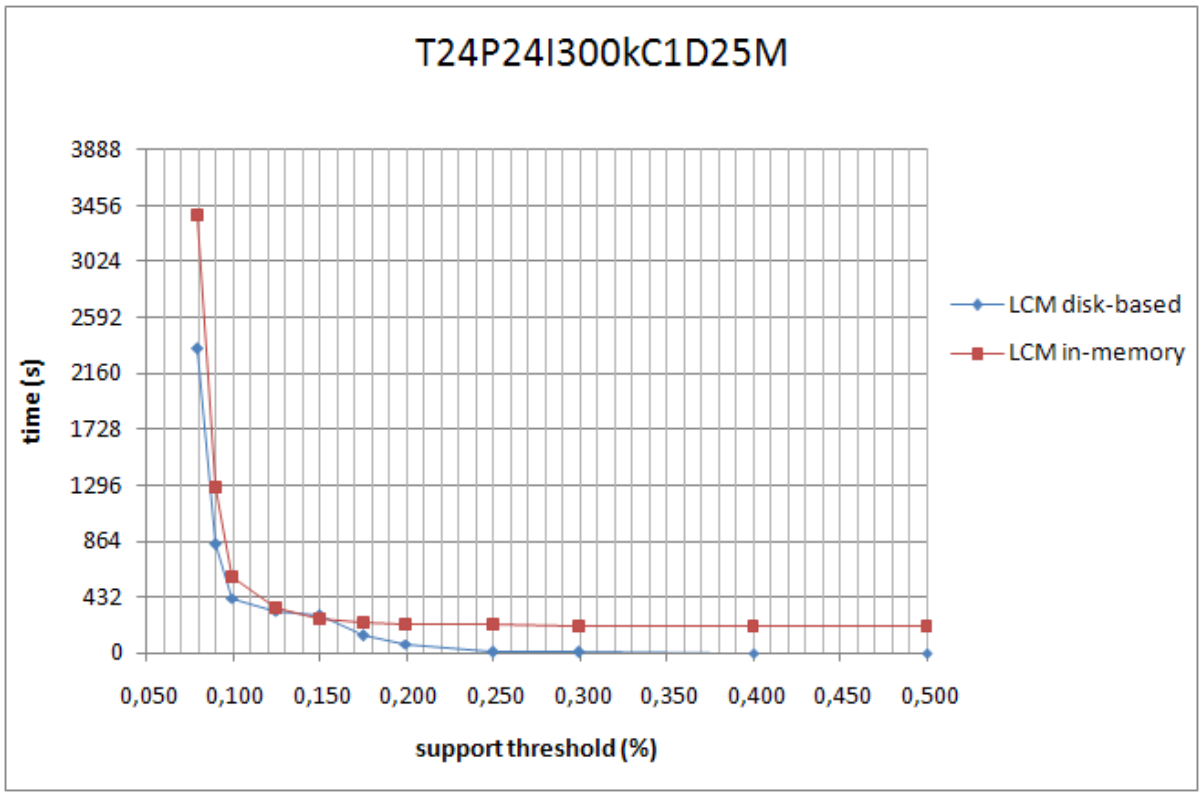

Figure 13. T24P24I300kC1D25M: LCM in-memory vs. disk-based

found for such support thresholds is too large to extract useful information, it is still interesting to notice the good results achieved by LCM disk-based on a dense dataset.

8.2 Page pre-faulting

As outlined in Chapter 5, the Hybrid-Tree environment exploits memory mapping to selectively load in main memory pages of the materialized tree. One of the benefits provided by the memory mapping mechanism is the so-called "lazy-loading".

This refers to the fact that, whenever a tree node is accessed during the mining process, only a small portion of the tree page where it resides – and not the entire page – is fetched from disk and loaded into main memory. The OS paging system is exploited to this aim, which makes use of small 4KByte pages. This feature results in a strictly on-demand data access scheme and saves considerable amounts of main memory.

For the purposes of the mining task, the real benefits of such a feature have been analysed. In particular, another data access scheme has been enforced during the mining process and performances of the two approaches have been compared.

The alternative data access scheme is based on a pre-faulting mechanism: whenever a node that is currently swapped out on disk needs to be accessed, the corresponding page is entirely loaded into main memory. The purpose is to minimize the number of distinct disk accesses and, for such events, maximize the sequentiality of the read data. The underlying assumption is that, if a page is accessed and one of its nodes is loaded in memory, other nodes included in the same page are also likely to be accessed in the future. This way, a large number of disk reads is avoided and the time spent with disk seeking operations is reduced.

8.2.1 Results

The page pre-faulting mechanism has been experimented on two datasets:

- T22P20I300kC0.75D20M
- T24P24I300kC1D25M

These datasets have been chosen as representatives because they illustrate the opposite effects pre-faulting can produce. Results are illustrated in Figure 14 and Figure 15.

For the first considered dataset, pre-faulting turns out beneficial in most cases. For the higher support thresholds much time is spent by the pre-faulted version with retrieving useless data, while the non-pre-faulted one only reads the small bits of data it actually needs. As a consequence, the pre-faulting mechanism does not produce an advantage. However, as the support threshold is decreased, the pre-faulted version gains a significant advantage. The assumption previously stated holds with this dataset.

In contrast, the pre-faulting mechanism has disastrous outcomes on the second considered dataset. Mining times are – sometimes significantly – higher than those obtained with no pre-faulting, for all support thresholds but one. This effect may be due to the data distribution: T24P24I300kC1D25M being a very dense dataset, few nodes are shared by a large number of transactions. In this case, loading all the surrounding nodes in main memory may be of no use and the mining process is forced to incur additional I/O costs with no actual benefit.

8.3 A medium-sized dataset

In this test case a medium-sized dataset, T22P20I250kC0.5D45M, has been considered. The purpose was to analyse the performance of LCM disk-based on a dataset whose size exceeds the available physical memory. Configuration I (see Section 7.1), which is only equipped with 2.5 Gbytes RAM, has been chosen to run this test, in order to "create" a memory bottleneck.

8.3.1 Results

Experimental results for this dataset are plotted in Figure 16 and Figure 17. In particular, Figure Figure 16 depicts mining times for the higher support thresholds. The same observations

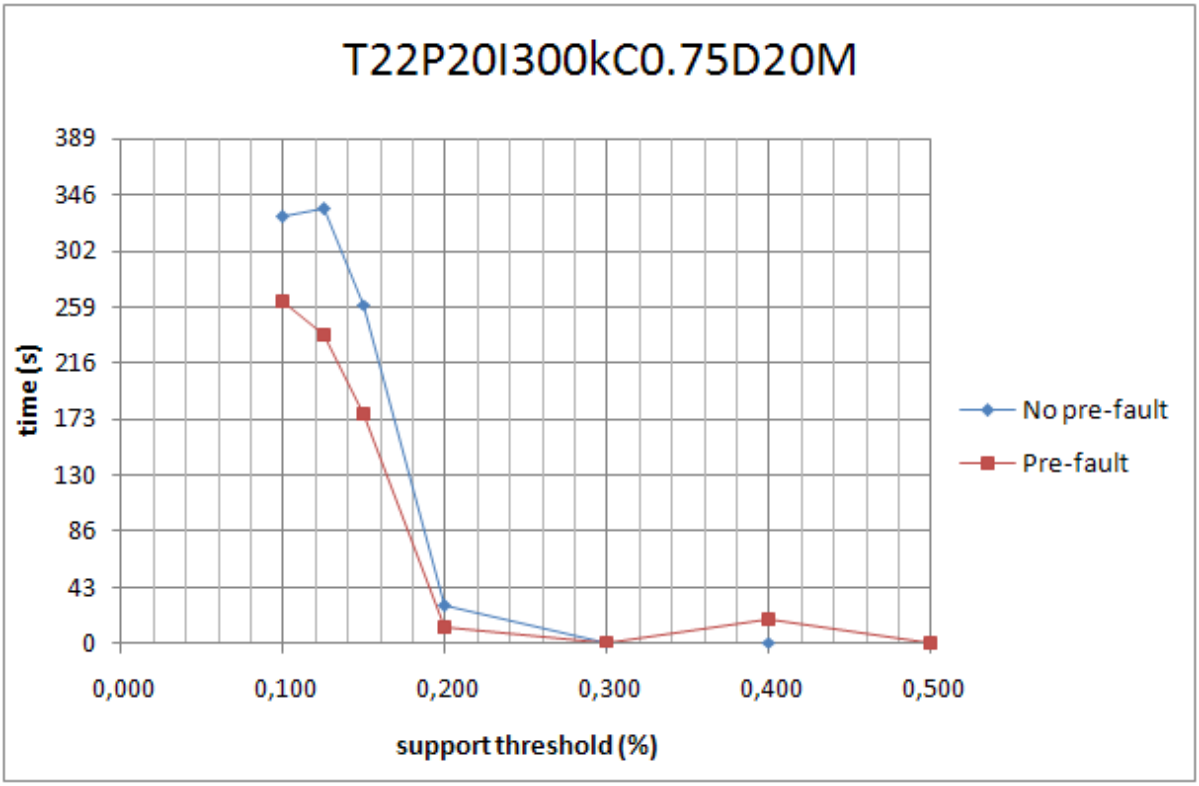

Figure 14. T22P20I300kC0.75D20M: LCM disk-based mining, impact of pre-faulting

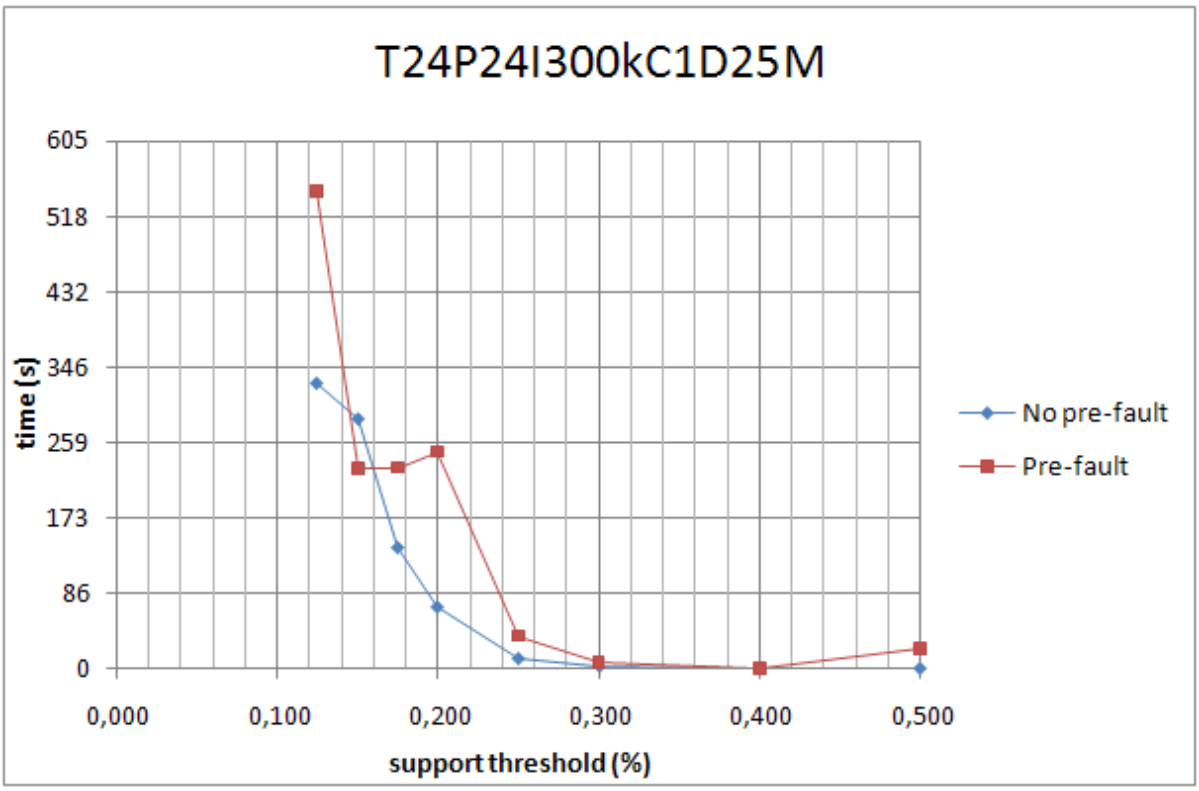

Figure 15. T24P24I300kC1D25M: LCM disk-based mining, impact of pre-faulting

stated for the T22P20I300kC0.75D20M dataset also hold for this dataset. From a certain threshold on, LCM in-memory starts outperforming the disk-based version.

However, because of the size of this dataset, lowering the support threshold gradually makes the mining process unmanageable by the memory-based algorithm. This situation is illustrated in Figure 17, where the memory-based version fails to mine at a support threshold of 0.1%. It has been interrupted after 15 hours because much swapping was occurring, while the actual CPU time only amounted to a few minutes. LCM disk-based, in contrast, still succeeds in the mining task, completing it after 5 hours. Such a result is far from being considered a timely one, but it still shows the better scalability of the disk-based approach.

8.4 Large datasets

In order to validate the proposed approach, experiments have been run on the following very large datasets:

- T20P18I150kC0.75D60M
- T20P18I150kC0.75D100M
- T20P18I150kC0.75D500M
- T20P18I150kC0.75D1000M

In an attempt to evaluate the scalability of the approach regardless of the considered data distribution, datasets have been generated with the same parameters but the number of transactions.

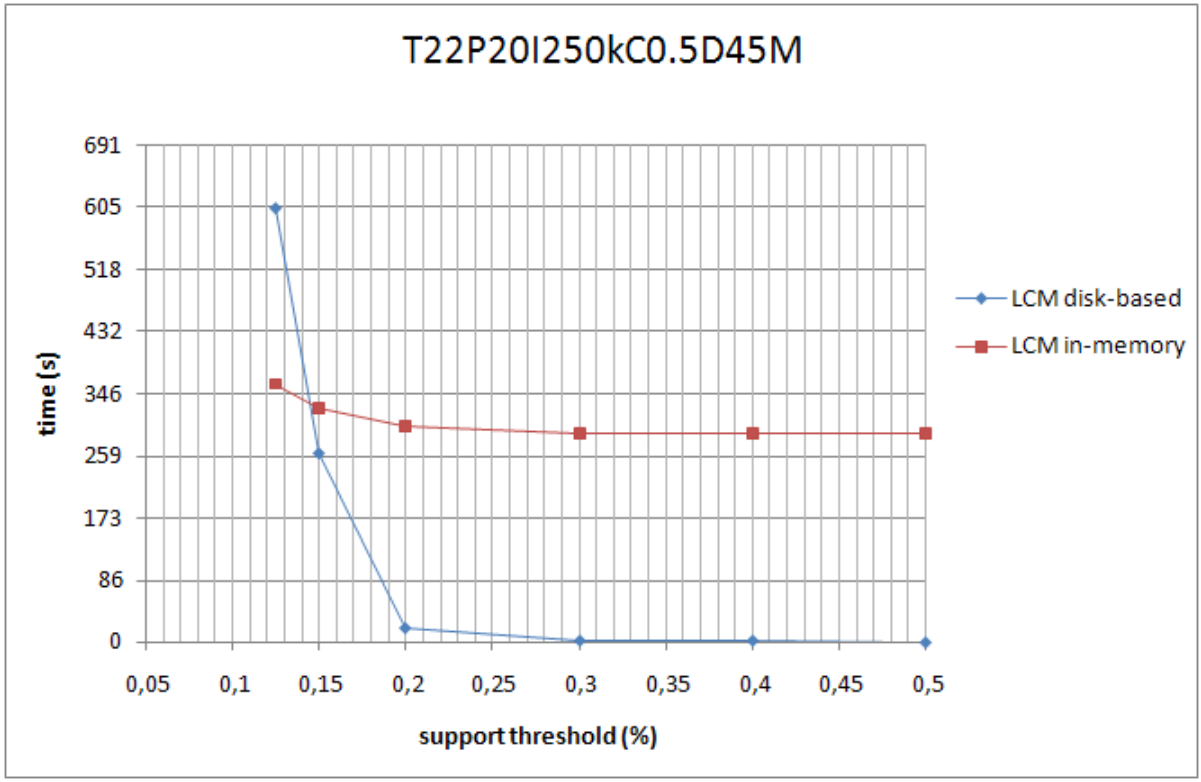

Figure 16. T22P20I250kC0.5D45M: LCM in-memory vs. disk-based, higher thresholds

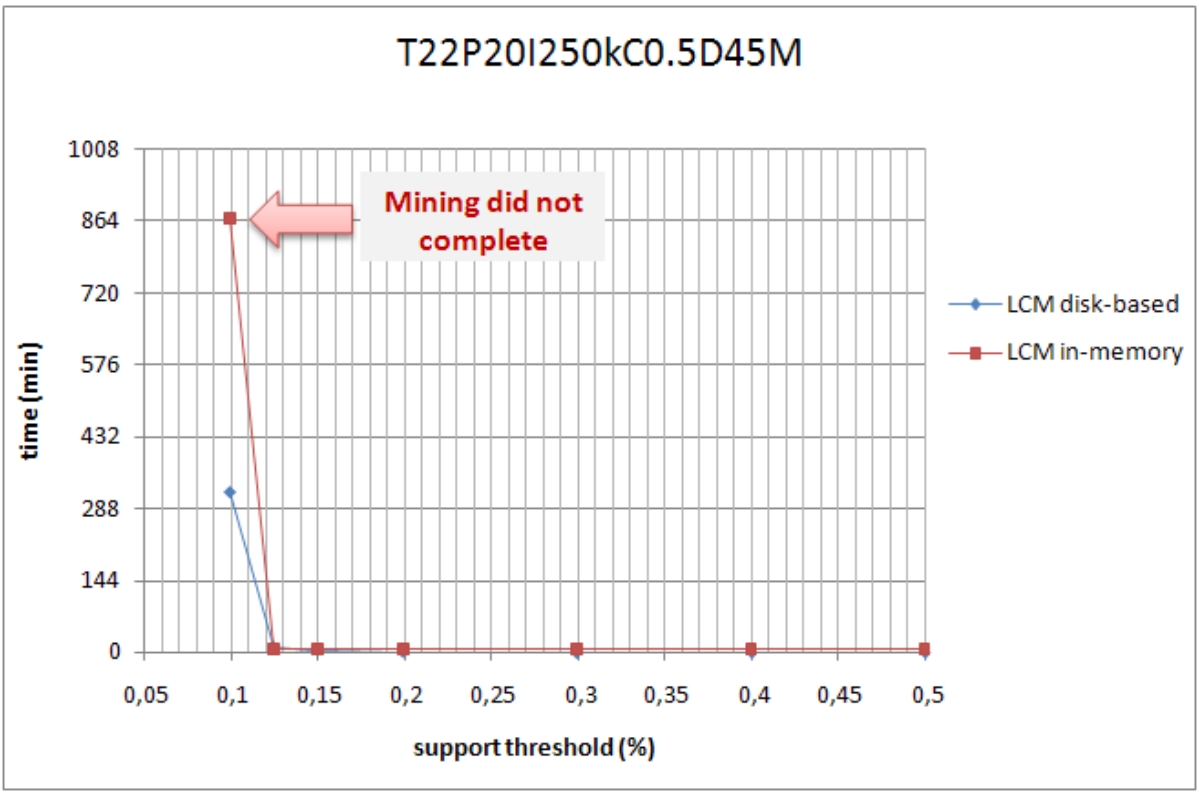

Figure 17. T22P20I250kC0.5D45M: LCM in-memory vs. disk-based, all thresholds

Comparison between the disk-based and the memory-based versions is no longer possible at this stage, because the memory-based version can only perform itemset extraction from these datasets at very high support thresholds in the best case and fails to allocate its memory-resident structures in the majority of cases.

8.4.1 Scalability

Results of the scalability test are plotted in Figure 18. Itemset extraction has been performed on the four aforementioned datasets at different support thresholds and performances have been compared.

As it appears from the plot, mining times for the first three datasets increase linearly with their sizes. This also true for the largest one billion transaction dataset when the higher support threshold are considered. However, as the support threshold decreases, mining times for this dataset still grow linearly, but following a different, less steep slope.

Once again, this may be due to the higher degree of compaction achieved on this dataset, which favors the mining process. Another reason may be identified with the smaller impact of I/O operations with respect to the extraction process itself, whose computational cost becomes predominant on such large datasets.

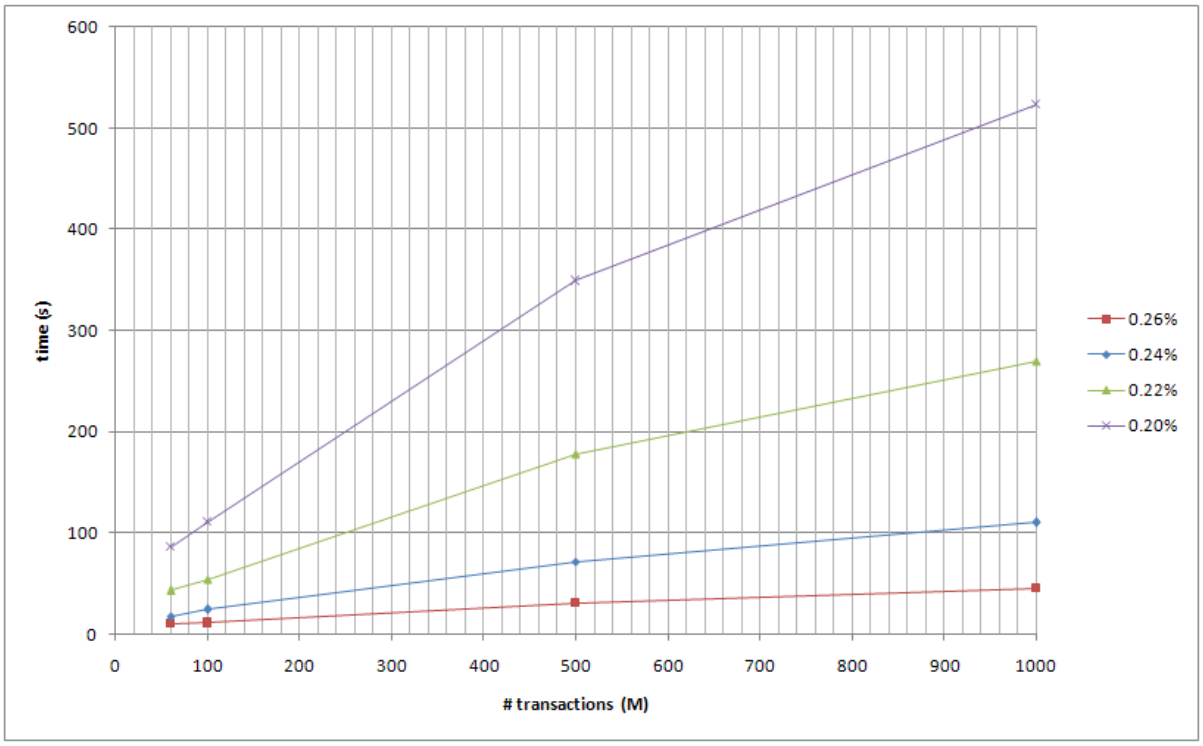

Figure 18. Scalability on large datasets

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

The solutions proposed in this master thesis work have proved able to successfully tackle frequent itemset extraction from much larger datasets than most other state-of-the-art approaches can handle. In addition, some improvements have been suggested and implemented to speed up the creation of the persistent tree structure.

Creating the persistent representation.

The main contribution to the materialization of the disk-based Hybrid-Tree is represented by development of a novel dataset sorting algorithm. Sorting transactions beforehand yields notable benefits in the creation phase, because it avoids multiple, temporally non-local visits of the same portion of the tree, thus positively impacting I/O times. However, this step was the most resource-demanding one and accounted for about 55%-60% of the overall creation time. The proposed technique splits the input dataset into a number of non-overlapping partitions, defining a partial order on the set of transactions. Each partition can then be sorted independently. This approach avoids several, costly passes over the dataset, thus saving on I/O costs; in addition, it scales linearly with the dataset size.

Index support.

To enhance mining activity, an indexing structure, called Item-Index, has been designed and developed for the Hybrid-Tree.

This compact index enables an item-based traversal of the Hybrid-Tree, required by several frequent itemset mining algorithms. The index consists of a per-item array-based structure, allowing fast loading and scanning. Although this index is not covering with respect to the mining task, the amount of replicated information (which includes composite pointers to nodes of the tree and their local support) is such that its final size can often exceed that of the Hybrid-Tree itself. Consequently, an ad-hoc compression technique has been developed which exploits a differential coding scheme and special structures maximizing the packing of data.

Further, some primitives have been developed to support the selective retrieval from the Hybrid-Tree of the transactional data explored during the mining process. These primitives heavily rely on the Item-Index to carry out the extraction of item-based projections of the dataset by efficiently navigating the tree in both top-down and bottom-up fashion. Their main advantage is that of enabling a tighter integration with mining algorithms.

Disk-based data mining.

Frequent itemset mining is carried out by integrating existing mining algorithms into our framework. Two main approaches are possible, involving different levels of complexity and providing different levels of scalability accordingly.

The first approach is the least invasive one; it consists in exploiting the Hybrid-Tree framework to extract a support-based projection from the dataset and subsequently mine it with the

algorithm of choice. The original algorithm can be directly chained to the extraction module and needs no adaptation or change of any kind. The scalability limit is thus represented by the size of the support-based projection, which needs to be memory-resident throughout the mining phase; when the dataset is huge, this solution may not suffice.

The second approach is likely to push this limit much farther. The underlying idea is to partition data loading and exploit the temporal locality of accesses of the algorithm. The target is to keep in memory, at any given time, only those portions of the dataset that are actually involved in the current mining activity. This approach requires a better understanding of the algorithm internals and, possibly, some modification of its routines. At the cost of an increased integration complexity, this solution can provide valuable benefits in terms of memory consumption. It has been applied to an efficient state-of-the-art mining algorithm, LCM v.2.

LCM v.2 is a mining algorithm which proved to outperform all other state-of-the-art mining algorithms in most cases; in addition, it has a linear complexity and is thus very suitable for large datasets. Yet, as a main drawback, its original implementation requires that the entire dataset be loaded in main memory, thus majorly limiting its scalability. Following the aforementioned partitioning approach, specific data access behaviors of this algorithm have been identified. Its main routine has been decomposed and interleaved with the selective retrieval of item-based projections from the Hybrid-Tree. The developed access primitives have been exploited to this aim. As a result, LCM has been enabled to perform itemset extraction by processing in main memory only a single item-based projection at a time. Its memory requirements have thus been greatly reduced, ultimately making it far more scalable.

Experimental results.

The experimental section of this master thesis work has been divided into two parts.

In the first set of experiments (i) the contribution proposed to improve the creation phase, namely the sorting algorithm, and (ii) the compression technique designed to reduce the final size of the Item-Index were validated upon five small-/medium-sized datasets (5M, 10M, 15M, 20M and 25M transactions) using a commodity PC and upon four large-sized datasets (45M, 60M, 100M and 500M transactions) using a mini-server machine; all datasets were synthesized with the IBM generator. The time complexity of the sorting algorithm showed linear when the different-sized datasets followed the same data distribution, and roughly linear otherwise. The devised compression technique achieved a size reduction of about 33% on the Item-Index of all the considered datasets; its time complexity is roughly linear in the number of nodes of the Hybrid-Tree.

The second series of experiments was aimed at assessing the mining performance of the integration proposed for the LCM v.2 algorithm. With small- and medium-sized datasets, the disk-based LCM outperforms the in-memory version for high support thresholds and is comparable to it for low support thresholds. This still holds as the dataset grows bigger and ultimately exceeds the size of the largest dataset manageable by the original memory-based approach on a fairly well-equipped machine. The mining process completes in a timely fashion (less than 45 minutes) on datasets larger than the physical memory by over one order of magnitude, while memory requirements are still very low (about 17% of the available memory).

A larger impact of I/O on mining times is to be expected as the available physical memory limit is approached. Although such an event has not occurred in our tests, this should not lead to a dramatical slow-down, thanks to the structured I/O framework which directly manages swapping.

9.2 Future work

This work proposed some techniques that proved able to enhance the performance and scalability of this disk-based data mining framework. Nonetheless, a number of issues could still be addressed, affecting both the materialization and the mining phase.

The disk-based tree is built through a gradual process which exploits several temporary structures stored on disk. Some of these structures are needed until the materialization has been fully created and therefore cannot be deleted prior to completion to save disk space. Among them is a temporary, uncompressed Item-Index. Because of the way it is obtained from the tree, it is currently unfeasible to compress it on-the-fly. It may be interesting, as a possible future development, to investigate alternative tree traversal strategies that enable to create full and disjoint Item-Index portions, which could then be directly compressed.

A further improvement may result from the study and development of new data structures to store the original dataset. The Hybrid-Tree currently exploits prefix-tree and array structures to provide a distribution-adaptive representation. As suggested in (16), bitmap structures are another effective and compact way to represent very dense portions of datasets; their integration in the Hybrid-Tree environment may lead to a higher compaction of the materialized tree and indirectly enhance the mining process by reducing memory consumption and I/O times.

Different interestingness constraints may also be integrated in the data mining framework. Constraint specification allows the analyst to better focus on interesting itemsets for the considered analysis task. Since the Hybrid-Tree provides a complete and compact representation of the transactional dataset, the item constraints proposed in (17) can be easily supported by the Hybrid-Tree. In addition, the proposed representation can be easily exploited by different algorithms that extract different and more compact itemsets (e.g., closed itemsets (18; 19) and maximal itemsets (20; 21)).

Finally, in furtherance of a more efficient LCM v.2 integration, two other improvements may be envisioned.

The first one concerns the amount of used memory. Because item-based projections are processed by LCM in main memory, the necessary memory-based structures must be afforded. In order to avoid reallocating memory for each projection, an estimate of the maximum amount of required memory is used to allocate them at the beginning of the mining session. However, this estimate is loose, which results in some waste. A tighter estimate could be suggested to reduce this waste.

As a second possible improvement, data retrieval and data mining routines may be parallelized, so that new item-based projections can be asynchronously fetched from disk while the CPU is busy with the current mining process. The benefit may be especially noticeable when mining is performed at low support thresholds, because a lot of CPU time is then required to process each item-based projection: this time could be profitably spent on filling a buffer, so as to reduce the chances of blocking I/O.

CITED LITERATURE

1. Han, J., Pei, J., and Yin, Y.: Mining frequent patterns without candidate generation. Proc. of ACM International Conference on Management of Data, 2000.

2. Rácz, B.: nonordfp: An fp-growth variation without rebuilding the fp-tree. 2nd Workshop on Frequent Itemset Mining Implementations (FIMI 04), 2004.

3. Uno, T., Kiyomi, M., and Arimura, H.: Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. 2nd Workshop on Frequent Itemset Mining Implementations (FIMI 04), 2004.

4. El-Hajj, M. and Zaiane, O. R.: Cofi-tree mining: A new approach to pattern growth with reduced candidacy generation. Proc. of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, November 2003.

5. Ye, F.-Y., Wang, J.-D., and Shao, B.-L.: New algorithm for mining frequent itemsets in sparse database. Proc. of the Fourth International Conference on Machine Learning and Cybernetics, pages 18 – 21, August 2005.

6. Savasere, A., Omiecinski, E., and Navathe, S.: An efficient algorithm for mining association rules in large databases. Proc. of the International Conference on Very Large Data Bases (VLDB), 1995.

7. Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., and Yang, D.: H-mine: Hyper-structure mining of frequent patterns in large databases. Proc. of the 2001 IEEE International Conference on Data Mining (ICDM 01), 2001.

8. Houtsma, M. and Swami, A.: Set-oriented mining of association rules. Proc. of the Eleventh International Conference on Data Engineering (ICDE), 1995.

9. Yong, Q.: Integrating frequent itemsets mining with relational database. Proc. of the 8th International Conference on Electronic Measurement and Instruments (ICEMI 07), 2007.

CITED LITERATURE (Continued)

10. Baralis, E., Cerquitelli, T., and Chiusano, S.: Index support for frequent itemset mining in a relational dbms. Proc. of the 21st International Conference on Data Engineering (ICDE 05), 2005.

11. Ramesh, G., Maniatty, W., and Zaki, M. J.: Indexing and data access methods for database mining. DMKD, 2002.

12. El-Hajj, M. and Zaiane, O. R.: Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. Proc. of the 2003 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August 2003.

13. Grahne, G. and Zhu, J.: Mining frequent itemsets from secondary memory. Proc. of the Fourth IEEE International Conference on Data Mining (ICDM 04), pages 91 – 98, 2004.

14. Buehrer, G., Parthasarathy, S., and Ghoting, A.: Out of core frequent pattern mining on a commodity pc. Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data mining, 2006.

15. Adnan, M. and Alhajj, R.: Drfp-tree: disk-resident frequent pattern tree. Applied Intelligence, 2007.

16. Stevens, W. R. and Rago, S. A.: Advanced Programming in the UNIX Environment. Addison Wesley Professional, 2005.

17. Uno, T., Kiyomi, M., and Arimura, H.: Lcm ver.3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. Proc. of the 1st international workshop on open source data mining (OSDM 05), 2005.

18. Pei, J., Han, J., and Lakshmanan, L. V. S.: Pushing convertible constraints in frequent itemset mining. Data Mining and Knowledge Discovery, 8(3):227 – 252, 2004.

19. Wang, J., Han, J., and Pei, J.: Closet+: searching for the best strategies for mining frequent closed itemsets. Proc. of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 03), pages 236 – 245, 2003.

20. Zaki, M. J. and Hsiao, C.-J.: Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Transactions on Knowledge and Data Engineering, 17(4):462 – 478, 2005.

CITED LITERATURE (Continued)

21. R. J. Bayardo, J.: Efficiently mining long patterns from databases. Proc. of the 1998 ACM SIGMOD international conference on Management of data (SIGMOD 98), pages 85 – 93, 1998.

22. Gouda, K. and Zaki, M. J.: Efficiently mining maximal frequent itemsets. Proc. of the 2001 IEEE International Conference on Data Mining (ICDM 01), pages 163 – 170, 2001.