Chapter 9
Integration of Dataset Scans in Processing Sets of Frequent Itemset Queries

Marek Wojciechowski, Maciej Zakrzewicz, and Pawel Boinski

Institute of Computing Science,
Poznan University of Technology,
ul. Piotrowo 2, 60-965 Poznan, Poland
{Marek.Wojciechowski,Maciej.Zakrzewicz,
Pawel.Boinski}@cs.put.poznan.pl

Abstract. Frequent itemset mining is often regarded as advanced querying where a user specifies the source dataset and pattern constraints using a given constraint model. In this chapter we address the problem of processing sets of frequent itemset queries, which brings the ideas of multiple-query optimization to the domain of data mining. The most attractive method of solving the problem with respect to possible practical applications is Common Counting which consists in concurrent execution of the queries using Apriori with the integration of scans of the parts of the database shared among the queries. The major advantage of Common Counting over its alternatives is its applicability to arbitrarily large batches of queries. If the memory structures of all the queries to be processed by Common Counting do not fit together in main memory, the set of queries has to be partitioned into subsets processed in several phases. We formalize the problem of dividing the set of queries for Common Counting as a specific case of hypergraph partitioning and provide a comprehensive overview of query set partitioning algorithms proposed so far.

1 Introduction

Frequent itemset discovery [1] is a very important data mining problem with numerous practical applications including market-basket analysis, medicine, telecommunications, and web usage analysis. Its goal is to discover the most frequently occurring subsets, called itemsets, in a database of sets of items, called transactions. Discovered frequent itemsets are often used to generate association rules. However, since generation of rules from itemsets is a rather straightforward, computationally inexpensive task, the focus of researchers has been mostly on optimizing the frequent itemset discovery process.

Many frequent itemset mining algorithms have been developed over the last two decades. The two most prominent classes of algorithms are determined by a strategy of traversing the pattern search space. Level-wise algorithms, represented by the classic Apriori algorithm [3], follow the breadth-first strategy, whereas pattern-growth methods, among which FP-growth [20] is the best known, perform the depth-first search.

Despite significant advances in frequent itemset mining, Apriori still remains the most widely implemented and used in practice frequent itemset mining algorithm due to its simplicity and satisfactory performance in real-world scenarios. Apriori starts with the discovery of 1-element frequent itemsets (i.e., frequent items), and then iteratively generates candidates (i.e., potentially frequent itemsets) from previously found smaller frequent itemsets and counts their occurrences in the database. To improve the efficiency of testing which candidates are contained in a transaction read from the database, the candidates are stored in a hash tree in main memory.

Frequent itemset mining is often regarded as advanced database querying where a user specifies the source dataset, the minimum support threshold, and optionally pattern constraints within a given constraint model [23]. A significant amount of research on efficient processing of frequent itemset queries has been done in recent years, focusing mainly on constraint handling [42][44][45][48][52] and reusing results of previous queries [6][12][35][39]. In terms of query optimization, the former can be regarded as optimizing individual queries separately, and the latter as optimizing sequences of queries [36].

This chapter is devoted to a relatively new problem of optimizing processing of sets of frequent itemset queries [51] that brings the concept of multiple-query optimization to the domain of frequent itemset mining. The idea is to process the queries concurrently rather than sequentially and take advantage of overlaps between queries' source datasets, although some of the proposed algorithms target also integration of in-memory data structures and computations. Sets of frequent itemset queries available for concurrent processing may arise in data mining systems operating in a batch mode or be collected within a given time window in multi-user interactive data mining environments. A motivating example from the domain of market basket analysis could be a set of queries discovering frequent itemsets from the overlapping parts of a database table containing customer transaction data from overlapping time periods.

Over the last decade we have proposed several approaches and algorithms to tackle the above problem. Here we focus on the most fundamental method called Common Counting [53], which basically consists in concurrent execution of the queries using Apriori with the integration of scans of parts of the database shared among the queries. Common Counting can be regarded as a fundamental solution of the problem of processing sets of frequent itemset queries for the two following reasons:

- methods proposed later for the Apriori algorithm were designed by extending Common Counting with further possibilities of computation sharing
- the first method of processing sets of frequent itemset queries dedicated to a newer FP-growth algorithm is a direct adaptation of Common Counting.

In general, we claim that Common Counting and its extensions have more practical importance that the methods for FP-growth since:

- despite positive evaluation of FP-growth and its variants in scientific literature, the majority of data mining systems available on the market still implement Apriori (often with some performance-oriented extensions and/or modifications with respect to the original formulation by Agrawal and Srikant).

- concurrent processing of frequent itemset queries requires storing the memory structures of several queries in main memory at the same time (with proposed space-preserving optimizations in some of the methods), which is problematic in case of FP-growth which basically stores a compressed version of the whole source dataset in main memory for each query.

On the other hand, despite the fact that its Apriori-oriented successors have been shown to outperform Common Counting if the queries' datasets significantly overlap, the original Common Counting method might still be preferable in practical implementations because:

- it has very little overhead and thus is the most predictable of all the proposed methods in a sense that it is beneficial for virtually any, even very small, overlapping among the queries' datasets
- it does not conflict with constraint-handling extensions proposed for Apriori
- contrary to its successors it has virtually unbounded scalability with respect to the number of queries.

While focusing on Common Counting, in this chapter we will particularly concentrate on the last of its properties listed above. Basic formulation of Common Counting [53] assumes that memory structures (i.e., hash trees) of all frequent itemset queries (i.e., concurrent Apriori executions) fit together in memory, which may not always be the case, at least for initial Apriori iterations. If the memory can hold only a subset of all queries, then it is necessary to partition the queries into subsets called phases and scan the database once for each phase. This observation leads to an interesting optimization problem with the goal of selecting from the set of all feasible partitionings the one (or one of) resulting in the minimal I/O cost of database scans [54].

In this chapter we summarize our research on query set partitioning algorithms for Common Counting by: (1) providing a formulation of the problem as a particular case of the hypergraph partitioning problem followed by a discussion regarding its computational complexity and obtaining the input information required by the partitioning algorithms; (2) reviewing all the proposed partitioning algorithms: CCRecursive [54], CCFull [57], CCCoarsening [58], CCAgglomerative [56], CCAgglomerativeNoise [8], CCGreedy, and CCSemiGreedy [10]; (3) presenting the extensive results of experiments aimed at evaluating the performance and accuracy of the algorithms.

2 Frequent Itemset Mining and Apriori Algorithm

In this section we review basic definitions concerning frequent itemset mining problem formulation as well as the classic Apriori algorithm, which is regarded as the basic algorithm for the Common Counting technique.

2.1 Basic Definitions and Problem Statement

Definition 1. Let I be a set of literals, called items. An *itemset* X is a set of items from I ($X \subseteq I$). The *size* of an itemset is the number of items in it. An itemset of size k is called a k-itemset. A *transaction* over I is a couple $T = \langle tid, X \rangle$, where *tid* is a transaction identifier and X is an itemset. A database D over I is a set of transactions over I such that each transaction has a unique identifier.

Definition 2. A transaction $T = \langle tid, X \rangle$ *supports* an itemset Y if $Y \subseteq X$. The *support* of an itemset Y in D is the number (or percentage) of transactions in D that support Y.

Definition 3. An itemset is called frequent in D if its support is no less than a given minimum support threshold.

Problem 1. Given a database D and a minimum support threshold *minsup*, the problem of *frequent itemset mining* consists in discovering all frequent itemsets in D together with their supports.

2.2 Algorithm Apriori

The Apriori algorithm for frequent itemset discovery is formally presented in Fig. 1. In the formulation of the algorithm, F_k denotes the set of all frequent k-itemsets, and C_k denotes a set of potentially frequent k-itemsets, called candidates.

```
Input: $D$, $minsup$
(1)  $F_1$ = frequent 1-itemsets
(2)  for ($k$=2; $F_{k-1} \neq \emptyset$; $k$++) do begin
(3)     $C_k = apriori\_gen(F_{k-1})$
(4)     forall transactions $t \in D$ do begin
(5)        $C_t = subset(C_k, t)$
(6)        forall candidates $c \in C_t$ do
(7)           $c.counter$++
(8)     end
(9)     $F_k = \{c \in C_k \mid c.counter \geq minsup\}$
(10) end
(11) Answer = $\bigcup_k F_k$
```

Fig. 1. Apriori

Apriori starts with the discovery of frequent 1-itemsets, i.e., frequent items (line 1). For this task, the first scan of the database is performed. Before making the k-th pass (for k>1), the algorithm generates the set of candidates C_k using F_{k-1} (line 3). The candidate generation procedure, denoted as *apriori_gen*(), provides efficient pruning of the search space, and will be described later. In the k-th database pass (lines 4-8), Apriori counts the supports of all the itemsets in C_k. (In practice, the database pass is performed only if the set of generated candidates is not empty.) The key step of this phase of the algorithm is determining which candidates from C_k are contained in a transaction t retrieved from the database. This step is denoted in the algorithm as a call to the *subset*() function that will be described later. At the end of the pass all itemsets in C_k with a support greater than or equal to the minimum support threshold *minsup* form the set of frequent k-itemsets F_k (line 9). The algorithm finishes work if there are no frequent itemsets found in a given iteration (condition in line 2) and returns all the frequent itemsets found (line 11).

The candidate generation procedure (*apriori_gen*() function in the algorithm) consists of two steps: the join step and the prune step. In the join step, each pair of frequent *k*-1-itemsets differing only in the last item (according to the lexicographical order of the items within itemsets) is joined to form a candidate. In the prune step, itemsets having at least one subset that was found infrequent in the previous Apriori iteration are removed from the set of candidates.

The key to the overall efficiency of the Apriori algorithm is efficient checking which candidates are contained in a given transaction (*subset*() function in the algorithm). In order to avoid costly testing of each candidate for inclusion in a transaction retrieved from the database, candidates are stored in a special in-memory data structure, called hash tree. Leaves of a hash tree contain pointers to candidates, while the root node and internal nodes contain hash tables with pointers to their child nodes. In order to check which candidates are stored in a given transaction, all the subsets of the transaction are pushed down the hash tree. Candidates pointed by the leaves that are reached in the tree traversal operation are then verified for actual inclusion in the transaction.

3 Frequent Itemset Queries – State of the Art

In this section we review the most important research directions regarding frequent itemset queries: languages and programming interfaces for data mining, optimizing single queries in the form of constraint-based mining, and optimizing sequences of queries by reusing results of previously executed queries.

3.1 Frequent Itemset Queries

The research on data mining queries[1] was initiated by the pioneering work of Imielinski and Mannila [23] who envisioned the evolution of data mining systems analogous to that of database systems. They claimed that one of the major forces behind the success of database management systems (DBMS) had been the development of query languages, SQL in particular. Firstly, SQL together with a relational database API resulted in decoupling applications from a database backend. Secondly, the ad hoc nature of querying posed a challenge to build general-purpose query optimizers. Based on the above observations, Imielinski and Mannila postulated that formulation of a data mining query language could become a foundation for the development of general purpose next-generation data mining systems, which they called Knowledge and Data Discovery Management Systems (KDDMS). Such systems would allow knowledge discovery from data as well as storing the discovered patterns, rules, models, etc.[2] in the database for further querying. Imielinski and Mannila introduced the term *inductive database* for a database that apart from data also stores discovered knowledge. This term has been subsequently used by some researchers (e.g., [36]) to

[1] Imielinski and Mannila used the term *KDD query*.

[2] Imielinski and Mannila used the term *KDD object* to describe results of data mining queries and considered three types of KDD objects: rules, classifiers, and clusterings.

describe the research area devoted to data mining query languages and data mining query optimization, with a particular focus on frequent itemset and association rule mining.

Several data mining query languages were proposed following the statement of direction provided by Imielinski and Mannila. In [11] the authors proposed to extend SQL with the MINERULE operator for extracting association rules from the database and storing them back in a database relation. The proposed extension of SQL supported the following features: selecting the source dataset for mining with a possibility of grouping normalized data into sets, definition of the required structure of discovered rules and constraints regarding them, and specifying support and confidence thresholds.

In [24][25] another extension of SQL, called MSQL, was proposed. The focus of the presented approach was not only on the data mining query language itself but also on the application programming interface through which MSQL queries would be send by applications to the data mining system. As for MSQL syntax, it was oriented on both discovering new rules and querying previously discovered rules stored in the database. In contrast to the approach from [11] which used standard SQL queries to check which data supported or violated a given rule, MSQL offered explicit language constructs for that purpose.

In [19] another data mining query language, called DMQL, was introduced as a basic interface to the DBMiner data mining system [18]. One striking difference between DMQL and the two languages mentioned earlier was a much broader scope of DMQL, which supported several other data mining techniques apart from association rule discovery, i.e., mining characteristic, classification, and discriminant rules. Another advantage of DMQL over MINERULE and MSQL was the direct support for incorporating background knowledge in the form of concept hierarchies (taxonomies) in order to discover generalized rules.

A few years after the first proposals of data mining query languages, MineSQL [38][39] was proposed as the first language supporting mining frequent itemsets as a final form of discovered knowledge. Two other types of patterns handled by MineSQL were association rules and sequential patterns. Borrowing the best features of its predecessors, MineSQL integrated SQL queries to select source datasets to be mined, supported creation of taxonomies and using them in the mining process, contained clauses to determine source data supporting or violating discovered rules, and allowed a user to materialize discovered knowledge in the database for further analyses. As for the latter, a novel approach was taken, allowing data mining queries to be defining queries of materialized views, thus introducing the concept of materialized data mining views.

Unfortunately, the aforementioned language proposals by the research community had no or little influence on standards or existing database management systems. The relevant data mining standards that include executing association rule/frequent itemset mining task within their scope are: Java Data Mining [31], OLE DB for Data Mining [41], and SQL/MM Data Mining [26]. All of them treat association rule mining as building a mining model, which can then be browsed/queried, with frequent itemsets regarded as a by-product or an intermediate step. Nevertheless, implicitly a frequent itemset query specifying the source dataset and constraints on frequent itemsets (derived from user-specified association rule constraints) is still executed, which makes the problem considered in this chapter still relevant. Recently, Oracle provided a strong argument supporting the research on frequent itemset queries by including a PL/SQL package for mining frequent itemsets as one of the standard packages starting from the version 10g of its database server [43]. The package allows frequent itemset queries to be formulated in pure SQL, which is an important signal to the research community. Firstly, it means that data mining queries are present in one of the market-leading DBMSs. Secondly, it clearly indicates that future research on data mining queries should focus on frequent itemset queries.

3.2 Constraint-Based Frequent Itemset Mining

Early research on frequent itemset and association rule query optimization focused on optimizing queries individually in the form of constraint-based mining. The idea was to incorporate user-specified constraints into mining algorithms in order to not only restrict the number of returned itemsets/rules (which can be done by post-processing) but also to reduce the execution time. The first approach to constraint-based frequent itemset mining was presented in [48]. Considered constraints had the form of a disjunctive normal form (DNF) with each disjunct stated that a certain item must or must not be present. Three Apriori-based algorithms were proposed, each of which applied a different modification to the candidate generation procedure.

In [42] Lakshmanan et al. considered more sophisticated constraints on frequent itemsets by allowing the constraints to refer to item attributes. The authors provided the first classification of constraints, identifying two important constraint properties: anti-monotonicity and succinctness. For all the considered constraint types a method of handling them within the Apriori framework was proposed, and formalized in the CAP algorithm.

Pei and Han [44][45] added monotonicity, previously considered in the context of correlated sets, to the two frequent itemset constraint properties identified by Lakshmanan et al. and then identified a broad class of constraints that do not exhibit any of the three properties but become monotone or anti-monotone if a certain order over the item domain is assumed. The new class of constraints was called convertible constraints. After completing the classification of constraints for frequent itemset mining, the authors showed that pattern-growth paradigm (represented by the FP-growth algorithm) is more suitable for constraint-based mining than the Apriori-based approach by providing guidelines on efficient handling of all four types of constraints within FP-growth.

In [52] a completely different method of handling constraints in frequent itemset mining, called dataset filtering, was presented. Instead of integrating the constraint-handling techniques into mining algorithms, the authors showed that certain constraints allow the query to be transformed into a query operating on the subset of the original query's input dataset that is equivalent in terms of the results. Although this approach is applicable to a small subset of constraints considered in frequent itemset mining, it has two important advantages. Firstly, it is independent of any particular frequent itemset mining algorithm. Secondly, it does not conflict with the constraint-handling techniques integrated into Apriori or FP-growth.

Recent works oriented on supporting frequent pattern mining in a query-oriented fashion suggest that, contrary to previous beliefs, pushing constraints down into the mining process in order to optimize processing of an individual query is not a good approach in terms of the overall system performance [15][22]. The key observation was that if pattern constraints are handled in a post-processing phase, then the system may materialize all the frequent patterns, not just those forming the final result of the query. Such an approach maximizes the chances of reusing materialized patterns by subsequent queries which typically is the most efficient way of answering a frequent pattern query.

3.3 Reusing Results of Previous Frequent Itemset Queries

The fact that mining results are often materialized in the database for further analyses and browsing raised a natural question whether and under what circumstances results of previous frequent itemset queries can be reused to improve the execution time of a new query. To answer the question several techniques were proposed that can be generally described as optimizing sequences of frequent itemset queries, in the sense that when processing a given query it is assumed that the results of previous queries are available.

The research on reusing results of previous queries actually started under the name of "incremental mining" before the term "data mining query" came into use. Cheung et al. considered the scenario when new data is added to the previously mined database, thus potentially making some of the previously frequent itemsets infrequent and vice versa [12]. They proposed an Apriori-based algorithm, called FUP, that used information about the support of previously frequent itemset to prune candidates in the process of mining the incremented database. In their subsequent work [13], the authors generalized their technique in the form of the FUP2 algorithm that was able to efficiently handle not only insertions but also deletions of data[3].

Both FUP and FUP2 require iterative scans of the whole input dataset in the same manner as basic Apriori. With the aim of reducing the cost of I/O activity in the process of incremental mining, Thomas et al. [49] proposed an incremental frequent itemset mining algorithm that required at most one scan of the whole input dataset, while still being able to handle both insertions and deletions in the original dataset. To achieve its goal, the algorithm required that not only frequent itemsets from the original dataset had to be available but also negative border of the set of frequent itemsets, i.e., itemsets that were Apriori candidates but turned out to be infrequent. The algorithm's I/O activity was concentrated on inserted/deleted data, and one scan of the whole dataset was needed only if the information about the supports of itemsets from the original mining result and its negative border together with the information obtained from inserted/deleted data was not sufficient to determine the support of all itemsets potentially frequent in the modified database.

Nag et al. [40] considered an environment where a number of users concurrently issue association rule queries. In order to improve the overall performance of such a system, they suggested caching itemsets (frequent ones and those forming the negative border) generated as a by-product of previous association rule queries and use them in the frequent itemset mining stages of upcoming association rule queries.

[3] In fact, FUP2 also handled updates of the input data, treating them as combinations of insertions and deletions.

Obviously, the approach taken actually resulted in optimizing sequences of frequent itemset queries. To facilitate itemset caching, the authors introduced the concept of a knowledge cache and proposed several algorithms for maintaining the cache as well as using its contents within the Apriori framework. As for differences among the queries in the context of which using the cache was beneficial, only differences in the support threshold were considered.

In [6] incremental refinement of association rule queries was considered in the context of the MINERULE operator from [11] by Baralis and Psaila. The authors postulated that a user is likely to refine their query a couple of times before obtaining the expected results. This observation was the motivation for studying syntactic differences between the queries that allow one query to be efficiently answered using the results of another query. Three relationships which occur between two association rule queries were identified: equivalence, inclusion, and dominance. Although, the approach concerned association rules, not frequent itemsets, it inspired subsequent works devoted to frequent itemsets as well.

Meo [35] continued the work of Baralis and Psaila on refinement of association rule queries in the context of the MINERULE operator by providing the ground for a query optimizer supporting the equivalence, inclusion, and dominance relationships. The author identified unique constraints and functional dependencies as elements of database management system functionality that could support such an optimizer and proposed extra intermediate data structures, called mining indices.

In [59] reusing results of previous frequent itemset queries stored in the form of materialized data mining views from [39]. Syntactic differences between MineSQL queries were analyzed and six scenarios of reusing the results of one query by another query were identified. These scenarios covered the techniques from [12], [40], and two of the three relationships from [6] adapted to frequent itemset queries.

The aforementioned methods of optimizing sequences of data mining queries can be regarded as preparing the ground for optimizing sets of data mining queries. The difference between all the above approaches and the problem studied in this chapter is the fact that they all deal with a sequence of queries arriving to the system and processed in a pre-defined order, while we are given a batch of queries at once. Obviously, applying arbitrary order on a set of queries turns it into a sequence, which means that all methods designed to deal with sequences of frequent itemset queries are automatically applicable to sets of frequent itemset queries as well. In fact, as we have shown in [37], it is possible to maximize the chance of one query reusing the results of another query by choosing appropriate ordering of the queries and/or introducing additional queries into the sequence. However, such an approach can be successfully applied just to a small fraction of cases that can be handled by a method designed with sets of queries in mind, like Common Counting to which this chapter is devoted.

4 Optimizing Sets of Frequent Itemset Queries

This section contains our formal, generic model of frequent itemset queries and presents the problem of optimizing sets of frequent queries, followed by a discussion on possible solutions and related work.

4.1 Basic Definitions

Definition 4. A *frequent itemset query* is a tuple $dmq = (R, a, \Sigma, \Phi, \beta)$, where R is a relation, a is a set-valued attribute of R, Σ is a condition involving the attributes of R (called a *data selection predicate*), Φ is a condition involving discovered itemsets (called a *pattern constraint*), and β is the minimum support threshold. The result of dmq is a set of itemsets discovered in $\pi_a \sigma_\Sigma R$, satisfying Φ, and having support $\geq \beta$ (π and σ denote projection and selection).

It should be noted that the assumption of using set-valued attributes to store transactions in a database relation does not influence the generality of the proposed query model. If input dataset is stored in the classic first normal form (1NF) where either each item occupies a separate row (so called transactional data format) or is represented by a binary flag in a dedicated column (so called relational format), it will be converted to the form of collection of sets in the process of reading the data from the database.

Our query model is general in the sense that we pose no restrictions on the form of data selection predicates, which we assume will be specified in pure SQL, and pattern constraints whose form and nature is irrelevant for our Common Counting method, which will be discussed later.

Example 1. Given the database relation $R_1(a_1, a_2)$, where a_2 is a set-valued attribute and a_1 is an attribute of integer type. The frequent itemset query $dmq_1 = (R_1, a_2,$ "$a_1 > 5$", "$|itemset| < 4$", 3%) describes the problem of discovering frequent itemsets in the set-valued attribute a_2 of the relation R_1. The frequent itemsets with support of at least 3% and size less than 4 items are discovered in the collection of records having $a_1 > 5$.

Definition 5. The set of *elementary data selection predicates* for a set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ is the smallest set $S = \{s_1, s_2, ..., s_k\}$ of data selection predicates over the relation R such that for each u, v ($u \neq v$) we have $\sigma_{su}R \cap \sigma_{sv}R = \emptyset$ and for each dmq_i there exist integers $a, b, ..., m$ such that $\sigma_{\Sigma i}R = \sigma_{sa}R \cup \sigma_{sb}R \cup .. \cup \sigma_{sm}R$.

The set of elementary[4] data selection predicates represents the partitioning of the database determined by overlapping of queries' datasets. Each partition contains transactions shared by exactly the same subset of queries and for each partition this subset of queries is different. Thus, the database partitions corresponding to elementary data selection predicates will be units of data subject to the optimization of the disk I/O cost.

[4] It should be noted that we use the term "elementary" to describe the property that data selection predicates of all the queries from a batch can be expressed as a disjunction of some of the elementary data selection predicates and at the same time this set cannot be reduced by combining predicates with disjunction so that this property still holds. Obviously, in general such elementary data selection predicates can be syntactically complex, i.e., having a form of simpler predicates combined using logical operators.

The set of elementary data selection predicates for a given set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ can be generated using the following procedure:

1) generate S as the set of all possible conjunctions of predicates from the set $\{\Sigma_1, \neg\Sigma_1, \Sigma_2, \neg\Sigma_2, \ldots, \Sigma_n, \neg\Sigma_n\}$ such that for each pair of predicates Σ_i, $\neg\Sigma_i$, exactly one of them is present in the conjunction
2) remove from S the conjunction $\neg\Sigma_1 \wedge \neg\Sigma_2 \wedge \ldots \wedge \neg\Sigma_n$
3) remove from S all predicates s such that $\sigma_s R=\emptyset$

The first step of the above procedure generates a formula for each subset of the set of queries that selects all the data shared only by this subset of queries. Obviously, the number of such formulas is 2^n. In the second step, the formula representing the empty subset of the set of queries (i.e., selecting data that do not belong to any query) is discarded. Finally, the formulas that select no data, i.e., do not correspond to any actual partition of data implied by the overlapping of queries' datasets are removed.

The last step of the procedure is the most challenging one. Some of the formulas will be discarded after syntactic analysis (i.e., self-contradictory formulas), while others will be identified as selecting no data only after the first execution of SQL queries corresponding to them. In the latter case, the execution of a set of frequent itemset queries will start with the superset of the actual set of elementary data selection predicates that will be tuned at early stages of the mining process. It should be noted that in scenarios that we believe are the most typical, i.e., involving all the queries selecting data according to the same attribute (e.g., queries mining data from different periods of time) syntactic analysis should be sufficient to eliminate redundant formulas from the set of elementary data selection predicates. Even if that is not the case, unnecessary SQL queries could be avoided thanks to statistics collected by the system. Nevertheless, we claim that generation of the set of elementary data selection predicates based on the queries' data selection predicates is a task for a SQL query optimizer.

Example 2. Given the relation $R_1(a_1, a_2)$ and three data mining queries: $dmq_1 = (R_1, a_2, \text{“}5 \le a_1 < 20\text{”}, \emptyset, 3\%)$, $dmq_2 = (R_1, a_2, \text{“}10 \le a_1 < 30\text{”}, \emptyset, 5\%)$, $dmq_3 = (R_1, a_2, \text{“}15 \le a_1 < 40\text{”}, \emptyset, 4\%)$. The set of elementary data selection predicates is then $S = \{s_1 = \text{“}5 \le a_1 < 10\text{”}, s_2 = \text{“}10 \le a_1 < 15\text{”}, s_3 = \text{“}15 \le a_1 < 20\text{”}, s_4 = \text{“}20 \le a_1 < 30\text{”}, s_5 = \text{“}30 \le a_1 < 40\text{”}\}$.

4.2 Problem Formulation

The problem of efficient processing of sets of frequent itemset queries could be formalized as the following optimization problem:

Problem 2. Given a set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, the problem of *multiple-query optimization* of DMQ consists in generating an algorithm to execute DMQ that minimizes the overall processing time.

The trouble with the above problem formulation is that a hypothetical multi-query optimizer for frequent itemset queries would need formulas for estimating the costs of various execution plans. Obviously, before such cost formulas for multi-query execution strategies could be developed, they had to exist for single queries[5], which unfortunately is not the case yet.

We claim that, taking the above observation into account, the present research on efficient processing sets of frequent itemset queries should focus on proposing algorithms that offer performance improvement over sequential execution, at least in typical scenarios, by sharing computations among the queries. Common Counting, presented in detail in the next section, is such a method.

4.3 Related Work on Multi-query Optimization

Multiple-query optimization has been extensively studied in the context of database systems (see [47] for an overview). The idea was to identify common subexpressions (selections, projections, joins, etc.) and construct a global execution plan minimizing the overall processing time by executing the common subexpressions only once for the set of queries [5] [28]. Many heuristic algorithms for multiple-query optimization in database systems were proposed (e.g., [46]). Data mining queries could also benefit from the general strategy of identifying and sharing common computations. However, due to their different nature they require novel multiple-query processing methods.

To the best of our knowledge, apart from the problem considered in this paper, multiple-query optimization for frequent pattern queries has been considered only in the context of frequent pattern mining on multiple datasets [30]. The idea was to reduce the common computations appearing in different complex queries, each of which compared the support of patterns in several disjoint datasets. This is fundamentally different from our problem, where each query refers to only one dataset and the queries' datasets overlap.

Earlier, the need for multiple-query optimization has been postulated in the somewhat related research area of inductive logic programming, where a technique based on similar ideas as Common Counting has been proposed, consisting in combining similar queries into query packs [7].

5 Common Counting

In this section we present the Common Counting method for efficient processing of sets of frequent itemset queries. We start with a basic algorithm that does not take into account the limit of the available memory, and then discuss the way of extending it to deal with such a practical restriction.

5.1 Basic Algorithm

The motivation for Common Counting is the observation that for a set of frequent itemset queries whose input datasets overlap, the most visible common operation in their Apriori-based execution is reading the shared parts of the database in the process of counting the candidates. Common Counting reduces the I/O costs with respect to sequential processing by concurrent execution of a set of frequent itemset queries

[5] For example, in order to estimate the cost of sequential execution of a set of queries.

using Apriori and integration of scans of the shared parts of the database. The pseudo-code of Common Counting is presented in Fig. 2.

```
Input: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, where $dmq_i = (R, a, \Sigma_i, \Phi_i, minsup_i)$
(1)  $S$ = set of elementary data selection predicates for $DMQ$
(2)  for ($i$=1; $i \leq n$; $i$++) do
(3)    $C_{1,i}$ = all possible 1-itemsets
(4)  for ($k$=1; $C_{k,1} \cup C_{k,2} \cup .. \cup C_{k,n} \neq \varnothing$; $k$++) do
(5)  begin
(6)    for each $s_j \in S$ do
(7)    begin
(8)      $CC = \{C_{k,i} : \sigma_{sj}R \subseteq \sigma_{\Sigma i}R\}$
(9)      if $CC \neq \varnothing$ then $count(CC, \sigma_{sj}R)$
(10)   end
(11)   for ($i$=1; $i \leq n$; $i$++) do
(12)   begin
(13)     $F_{k,i} = \{c \in C_{k,i} : c.counter \geq minsup_i\}$
(14)     $C_{k+1,i} = apriori\_gen(F_{k,i})$
(15)   end
(16) end
(17) for ($i$=1; $i \leq n$; $i$++) do
(18)   Answer$_i$ = $\sigma_{\Phi i} \bigcup_k F_{k,i}$
```

Fig. 2. Common Counting

The initial step of Common Counting is the generation of the set of elementary data selection predicates for the set of queries as discussed in Sect. 4.1 (line 1). After that, Common Counting iteratively generates and counts candidates for all frequent itemset queries. In the first iteration, for all the queries, the set of candidates is the set of all possible items (lines 2-3). The candidates of the size k (k>1) are generated from frequent itemsets of size k-1, separately for each query (lines 11-15). Generation of candidates (represented in the pseudo-code by the *apriori_gen*() function) is performed exactly as in the original Apriori algorithm. The candidates generated for each query are stored in a separate hash tree. The iterative process of candidate generation and counting ends when for all the queries no further candidates can be generated (the condition in line 4).

Occurrences of candidates for all the queries are counted during one integrated database scan in the following manner. For each elementary data selection predicate, the transactions from its corresponding database partition are read one by one. For each transaction the candidates of the queries referring to the database partition being read are considered, and the counters of candidates contained in the transaction are incremented (lines 6-10). The inclusion test is performed by confronting the transaction with hash trees of all the queries referring to the database partition containing the transaction. Candidate counting is represented in the pseudo-code as the *count*() function. It should be noted that if a given elementary data selection predicate is shared by several queries, its corresponding database partition is read only once during each candidate counting phase.

The formulation of Common Counting from Fig. 2 does not incorporate pattern constraints into the actual mining process, leaving them for post-processing (line 18). The reason for this is the fact that the optimization applied by Common Counting concerns only database access. Nevertheless, it should be noted that the Common Counting scheme does not interfere in any way with constraint-handling techniques described in sect. 3.2, meaning that these techniques could be incorporated into Common Counting in the same way they are incorporated into pure Apriori.

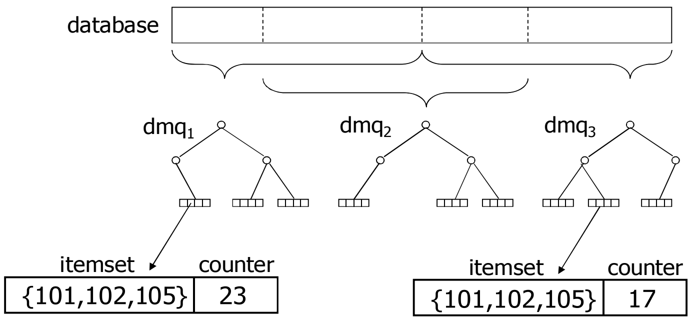

Fig. 3. Illustration of Common Counting and its memory structures

The idea of Common Counting and its memory structures are illustrated in Fig. 3 for the set of three queries. Each query creates its own hash tree to store its candidates. If a given itemset is generated as a candidate by more than one query, it appears in more than one hash tree. Clearly, there are more possibilities of computation sharing among the queries beyond just integrating scans of input data. However, such a tighter integration comes at a certain price, which will be briefly described in Sect. 9.

Common Counting was designed with the assumption that partitions of the database corresponding to elementary data selection predicates can be efficiently retrieved, for example using indexes. However, there is no guarantee that for any data selection predicate specified in a user's query an appropriate index will be present in the database. Interestingly, Common Counting can be even more beneficial if full scans of the database relation containing input datasets of the queries are necessary to retrieve the data partitions. However, a change of the way Common Counting reads data partitions is required to adapt it to the absence of more efficient data access paths than full table scan. Instead of reading partitions one by one, one scan of the whole relation should be performed and for each transaction the check to which queries it belongs should be performed. Such an approach is possible thanks to the fact that Common Counting actually does not require that partitions are read as a whole and can switch from partition to partition during the scan of the database relation. Obviously, after the above modification Common Counting will have to read more data than it would have to if an appropriate index was available. Nevertheless, if full table scans are the only option, sequential execution of the queries will have its performance relatively more degraded than Common Counting, since for each query a full scan will be needed.

5.2 Motivation for Query Set Partitioning

The basic Common Counting assumes that in each of its iterations candidate hash trees of all the frequent itemset queries forming a batch can reside in the main memory at the same time, and thus only one database scan is needed to count current candidates of all the queries. Obviously, in practice memory is going to be limited, so that simple strategy may not always be applicable.

Actual memory requirements of the Common Counting method depend on the number of queries, their support thresholds, and characteristics of the database. Nevertheless, in order to make Common Counting applicable in practice for arbitrarily large batches of queries, regardless of their predicates and the nature of the database, a solution enabling counting the candidates stored in hash trees whose total size exceeds the memory limit has to be provided.

To address the above issue, we propose to partition the set of queries into subsets so that the hash trees of the queries from each subset fit into memory. After the partitioning, counting of the candidates will be performed in several phases, with each of the resulting subsets of queries having their candidates counted during one database scan.

Clearly, the I/O cost of a Common Counting iteration divided into phases will be greater than it would be if query partitioning was not necessary. However, this cost will still be smaller than in case of sequential execution of the queries because the data sharing among the queries assigned to the same phase will still be taken advantage of. It should be noted that in general for a given set of queries many different partitioning will be possible, resulting in potentially different I/O costs. This observation leads to an interesting optimization problem of choosing the partitioning with minimal resulting I/O costs. Before we formalize the problem and present algorithm to solve it, we will discuss a few key issues that query partitioning algorithms for Common Counting have to take into account.

5.3 Key Issues Regarding Query Set Partitioning

In order to be able to verify if a given assignment is feasible and compare feasible assignments in terms of resulting I/O costs, the query set partitioning algorithm has to be provided with the sizes of hash trees and the sizes of database partitions corresponding to the elementary data selecting predicates representing data sharing among the queries.

Since the sizes of candidate hash-trees change between Apriori iterations, the assignment of queries to Common Counting phases has to be performed at the beginning of every Apriori iteration. A partitioning algorithm requires that sizes of candidate hash-trees are known in advance. Therefore, in each iteration of Common Counting, we first generate all the candidate hash trees, measure their sizes, save them to disk, partition the data mining queries into phases, and then load the hash trees from disk when they are needed during Common Counting phases. We have also considered estimating hash tree sizes in order to avoid the costs of migrating pre-created hash trees between main memory and disk [9]. The estimation formula was designed with a tendency to overestimate the size of a tree so as to minimize the chance that some of the phases actually do not fit into memory. The negative side of the approach taken was that the partitionings based on estimates were significantly worse in terms of I/O costs than those based on actual computed sizes. Therefore, we consider generating all the candidate hash trees in advance and swapping some of them to disk if there is not enough memory to keep them all together as a primary option.

As for the sizes of partitions of the database determined by overlapping of the queries' datasets, it is true that they are not known before the first iteration of Common Counting. Fortunately, the sizes of the database partitions will not be required until the second Common Counting iteration, when the hash trees are starting to be constructed. In the first iteration candidates for all the queries are all single items from the database and storing their counters for a sensible number of queries should not be a problem. Therefore, only one scan of the database will be needed in the first Common Counting iteration with no assignment of queries to phases, i.e., with one phase including all the queries. During that scan the actual sizes of database partitions can be calculated, so we can assume that they are available for subsequent Common Counting iterations.

The exhaustive search for an optimal assignment of queries to Common Counting phases is inapplicable for large batches of queries due to the size of the search space (expressed by a Bell number). Moreover, as we will show in the next section, the problem is NP-hard. Therefore, in practice heuristic algorithms have to be used.

Finally, let us consider the effect of available access paths to data partitions on the problem of assigning queries to Common Counting phase. The above discussion and the problem formulation from the next section are valid if the selective access to partitions of the relation with input data is possible, which we regard as the primary scenario. If the only access path to the data is full table scan, then Common Counting will be reading the whole relation in each Common Counting phase. Thus, the data sharing among the queries assigned to the same will be irrelevant, and only the number of phases will be important. Consequently, the problem will become a classic bin packing problem, where a number of objects (in our case – hash trees) are to be packed to bins of a given size (in our case – memory limit) so that the number of bins (in our case – phases) is minimal. Although bin packing is an NP-hard problem, numerous efficient heuristics are known for solving it. Therefore, we will not analyze this scenario any further.

6 Frequent Itemset Query Set Partitioning by Hypergraph Partitioning

In this section we introduce the concept of data sharing hypergraph as a model of data sharing between frequent itemset queries. Then, in the context of data sharing hypergraph we formulate our problem of partitioning the set of frequent itemset queries as a particular case of hypergraph partitioning. Finally, we discuss computational complexity of the problem and review related work on hypergraph partitioning techniques.

6.1 Data Sharing Hypergraph

A set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ can be modeled as a weighted hypergraph whose vertices represent queries and hyperedges represent elementary data selection predicates. A hyperedge in the hypergraph corresponds to a database partition and connects the queries whose source datasets *share* that partition. Below we formally define a data sharing hypergraph in the context of elementary data selection predicates.

Definition 6. A *data sharing hypergraph* for the set of data mining queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ and its corresponding set of elementary data selection predicates $S = \{s_1, s_2, ..., s_k\}$ is a hypergraph $DSH=(V,E)$, where $V=DMQ$, $E=S$, and a vertex $dmq_i \in DMQ$ is incident to an hyperedge $s_j \in S$ iff $\sigma_{sj}R \subseteq \sigma_{\Sigma i}R$. Each vertex dmq_i has an associated weight $w(dmq_i)$ representing the amount of memory consumed by data structures of the query dmq_i. Each hyperedge s_j has an associated weight $w(s_j)$ representing the size of the database partition returned by the elementary data selection predicate s_j.

Note that the above definition of a data sharing hypergraph allows hyperedges incident to only one vertex in order to represent database partitions read by only one query. These hyperedges are necessary for a data sharing hypergraph to provide complete information required by the main Common Counting scheme, and will also be used to evaluate the partitioning objective in our hypergraph partitioning problem.

Example 3. Given three frequent itemset queries operating on the relation $R_1 = (a_1, a_2)$: $dmq_1=(R_1$, "a_2", "$5 \le a_1 <20$", $\emptyset$, 3%), $dmq_2=(R_1$, "a_2", "$10 \le a_1 <30$", $\emptyset$, 5%), $dmq_3=(R_1$, "a_2", "$15 \le a_1 <40$", $\emptyset$, 4%). The set of elementary data selection predicates for the set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, dmq_3\}$ is $S=\{$"$5 \le a_1 < 10$", "$10 \le a_1 <15$", "$15 \le a_1 < 20$", "$20 \le a_1 < 30$", "$30 \le a_1 < 40$"$\}$. The data sharing hypergraph for DMQ is shown in Fig. 4.

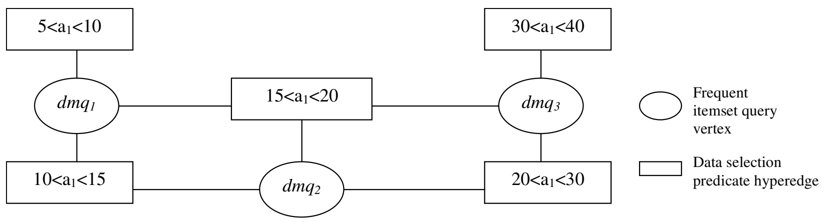
5<a1<10
30<a1<40
dmq1
15<a1<20
dmq3
Frequent itemset query vertex
10<a1<15
dmq2
20<a1<30
Data selection predicate hyperedge

Fig. 4. Example data sharing hypergraph

6.2 Hypergraph Partitioning Problem Formulation

The goal of query set partitioning for Common Counting is assigning queries to phases fitting into main memory in a way minimizing the overall I/O cost. Each of the phases returned by the partitioning algorithm is a set of frequent itemset queries for which a data sharing hypergraph can be constructed. Thus, query partitioning for Common Counting can be interpreted as a particular case of hypergraph partitioning.

After partitioning, elementary data selection predicates corresponding to database partitions shared by queries that have been assigned to different phases will be represented as hyperedges in more than one resulting hypergraph. In other words, a hyperedge that is cut by the partitioning will be partitioned into a number of hyper-edges connecting subsets of vertices previously connected by the original hyperedge. This is crucial for our problem because we need to preserve the information about data partitions to be scanned in each of the Common Counting phases.

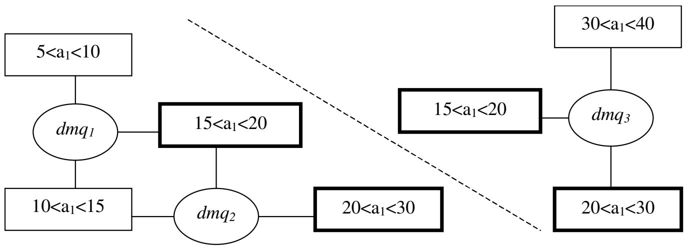

Fig. 5. Example partitioning of the data sharing hypergraph from Fig. 4

One of the possible partitionings of the data sharing hypergraph from Fig. 4, representing partitioning the set of queries into two phases is shown in Fig. 5. Hyper-edges that have been cut (partitioned) are presented in bold.

In terms of hypergraph partitioning, the goal of query partitioning for Common Counting can be stated as follows:

Problem 3. Given a data sharing hypergraph for the set of frequent itemsets queries $DSH = (V,E)$ and the amount of available main memory *MEMSIZE*, the goal is to partition the vertices of the hypergraph into k disjoint subsets $V_1, V_2, \ldots, V_k$, and their corresponding data sharing hypergraphs $DSH_1 = (V_1,E_1)$, $DSH_2 = (V_2,E_2)$, …, $DSH_k = (V_k,E_k)$ such that

$$\underset{x=1..k}{\forall} \sum_{dmq_i \in V_x} w(dmq_i) \leq MEMSIZE$$

minimizing

$$\sum_{x=1..k} \sum_{s_j \in E_x} w(s_j).$$

In the above formulation, the partitioning constraint has the form of an upper bound on the sum of weights of vertices in each partition, reflecting the amount of available memory, while the partitioning objective is to minimize the total sum of weights of hyperedges across all the partitions, representing the overall I/O cost of the Common Counting iteration. According to the classification from [32], the partitioning objec-tive in our problem formulation is equivalent to minimizing the k-1 metric, where the goal is to minimize the size of the hyperedge cut to which each cut hyperedge contributes *k*-1 times its weight (in the definition of the *k*-1 metric *k* denotes the number of partitions across which a cut hyperedge spans, not the total number of resulting partitions).

It should be noted that the number of resulting partitions (i.e., Common Counting phases) is not known a priori, and there is no lower bound on the sum of weights of vertices in each partition. Informally, the latter means that we do not require that the resulting partitions are of similar sizes.

6.3 Computation Complexity of the Problem

Our hypergraph partitioning problem is NP-hard since if we consider only hypergraphs with hyperedges connecting exactly two vertices, its decision version will restrict itself to the classical graph partitioning problem formulation from [14] (proof of NP-completeness by restriction). Taking that into account, for large number of vertices (frequent itemset queries) heuristic approaches have to be applied to solve the problem, resulting in possibly suboptimal solutions.

6.4 Related Work on Hypergraph Partitioning

Hypergraph partitioning has been extensively studied particularly in the domain of VLSI design [4]. In data mining context it has been proposed as a clustering technique in [34]. Many formulations of the hypergraph partitioning problem have been considered, differing in partitioning constraints and objectives (see e.g. [4] or [32]). Our formulation differs from typical approaches because:

- we do not have any balance constraint on the sizes of resulting partitions, only a strict upper bound on the sum of weights of vertices in a partition, reflecting the memory limit
- we do not specify the desired number of partitions in advance; in fact, the resulting number of partitions (phases) is irrelevant, only the partitioning objective matters
- for a hyperedge that is cut by the partitioning, we take into account the number of partitions to which the vertices connected by the cut hyperedge belong[6].

7 Query Set Partitioning Algorithms

This section presents heuristic algorithms that we proposed to solve the hypergraph partitioning problem formulated in the previous section. We have taken two different approaches to designing query set partitioning algorithms for Common Counting. The first was to invent new methods, dedicated to our particular problem. The algorithms designed this way are CCRecursive, CCFull, CCCoarsening, CCAgglomerative, and CCAgglomerativeNoise. An alternative approach was to apply some of the

[6] In other application domains it is more common just to check whether a hyperedge is cut or not. Nevertheless, the same approach as ours has also been considered in the VLSI domain.

well-known metaheuristics. Motivated by the reported success of application of the greedy approach to a related problem of k-way graph partitioning [27], we implemented a greedy and semi-greedy strategies in the context of our hypergraph partitioning problem. The resulting algorithms have been named CCGreedy and CCSemi-Greedy respectively.

7.1 CCRecursive

The CCRecursive algorithm directly utilizes the information contained in the data sharing hypergraph. Obviously, in order to minimize the partitioning criterion, the queries sharing an elementary data selection predicate should be assigned to the same phase. Since, in general it may not be possible for all the predicates, CCRecursive gives preference to predicates corresponding to larger data partitions.

```
$Phases = \{\emptyset\}$
sort $S = <s_1, s_2, ..., s_k>$ in descending order with respect to $cost(s_i)$
call $CCRecursive(S, DMQ, Phases)$

CCRecursive(S, DMQ, Phases):
begin
  ignore in S those predicates that are used by less than two dmqs
  for each $s_i$ in $S$ do begin
    $tmpDMQ = \{dmq_j \mid dmq_j = (R, a, \Sigma_j, \Phi_j, \beta_j), s_i \subseteq \Sigma_j, dmq_j \in DMQ\}$
    $commonPhases = \{p \in Phases \mid p \cap tmpDMQ \neq \emptyset\}$
    if $commonPhases = \emptyset$ then
      $newPhase = tmpDMQ$
    else
      $newPhase = tmpDMQ \cup \bigcup p \mid p \in commonPhases$
    end if;
    if $treesize(newPhase) \leq MEMSIZE$ then
      $Phases = Phases \setminus commonPhases$
      $Phases = Phases \cup newPhase$
    else
        $Phases = CCRecursive(<s_{i+1}, \ldots, s_k>, newPhase, Phases)$
    end if
  end
  add phase for each unassigned query
  compress Phases containing queries from DMQ
  return Phases
end
```

Fig. 6. CCRecursive

The detailed structure of the CCRecursive algorithm is given in Fig. 6. The algorithm iterates over all the elementary data selection predicates, sorted in descending order with respect to their I/O costs. For each elementary data selection predicate we identify all the data mining queries that include the predicate. If none of the identified queries has been already assigned, then we create a new phase (partition) and we put all the queries into the new phase. Otherwise, we merge the phases to which the assigned queries belonged and we assign the other queries to this new phase. If the size of the newly created phase exceeds the memory limit, then the phase is split into smaller ones by recursive execution of the algorithm with the list of data selection predicates reduced to only those following the predicate that led to exceeding the memory limit. (The auxiliary function *treesize*(Q), where Q is a set of data mining queries, represents total memory size required to hold candidate hash trees for all the queries in Q.)

At the end of the algorithm, we perform *phase compression*, which consists in reducing the number of phases by merging the phases so that the resulting phases still do not exceed the memory limit. This in fact leads to a bin packing problem which itself is an NP-hard problem. If the number of phases to compress prevents the exhaustive search for better packing, one of the heuristics proposed for the bin packing problem can be applied.

Although in general, in the context of our problem merging phases that do not share a database partition makes no sense in terms of minimizing the partitioning criterion, this step is required in CCRecursive since the phases may have been sharing a database partition corresponding to a data selection predicate removed in the recursive call of the algorithm.

7.2 CCFull

The problem with CCRecursive is that it does not take into account the fact that the queries may share more than one database partition. As a consequence, CCRecursive may not actually give precedence to query grouping/phase merging resulting in bigger gains with respect to the partitioning criterion. CCFull addresses this problem by considering the actual gains thanks to assigning a given subset of queries to the same phase. Another advantage of CCFull over CCRecursive is that CCFull does not require recursive calls, which makes its number of operations more predictable.

The first step of CCFull is generation of a *gain hypergraph* for the set of data mining queries. The gain hypergraph is a full (i.e., complete) hypergraph, in which vertices represent the data mining queries while hyperedges are labeled with weights which represent the amount of I/O cost reduction to be achieved if data mining queries connected with the hyperedge were executed together (in the same phase). If common execution of given data mining queries results in no reduction of I/O cost, the weight of the connecting hyperedge is zero. A sample gain hypergraph is shown in Fig. 7. For example, it can be noticed that common execution of the data mining queries dmq_0, dmq_2, and dmq_3 would reduce the total I/O cost by 16 units (the weight of the connecting hyperedge) compared with the sequential execution. Using the same example, it can also be noticed, that common execution of only the data mining queries dmq_1 and dmq_2 provides no cost reduction (the weight of the connecting hyperedge is zero).

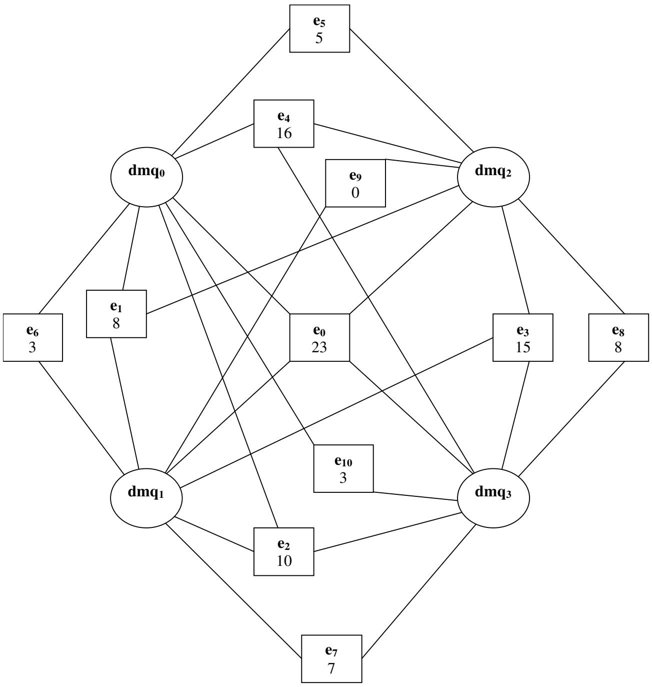

Fig. 7. Sample gain hypergraph

The gain hypergraph can be generated using the algorithm GenerateGainHypergraph shown in Fig. 8. The algorithm takes two arguments: the set of all elementary data selection predicates and the set of all data mining queries. First, the algorithm builds a full hypergraph whose nodes are the data mining queries. Each hyperedge receives the initial weight of zero. Then, for each hyperedge *e*, we create a set *P* of distinct data selection formulas involved in all data mining queries connected with the hyperedge *e*. I/O costs for executing the distinct data selection formulas from *P* are then summarized and the result is assigned to the hyperedge *e* weight.

```
GenerateGainHypergraph(S, DMQ):
begin
  generate a full hypergraph $G=\{V,E\}$, $V=DMQ$
  for each $e \in E$ do begin
    $e.gain = 0$
    $P = \{s_i \in S \mid \exists dmq_j \in e, dmq_j = (R, a, \Sigma_j, \Phi_j, \beta_j), s_i \subseteq \Sigma_j\}$
    for each $s \in P$ do begin
      $e.gain \mathrel{+}= cost(s) * (|\{dmq_j: dmq_j \in e, dmq_j = (R, a, \Sigma_j, \Phi_j, \beta_j), s_i \subseteq \Sigma_j\}| - 1)$
    end
  end
  return G
end
```

Fig. 8. Gain hypergraph generation algorithm

After having created the gain hypergraph, CCFull performs the following steps. All hyperedges are sorted in descending order according to their weights. Next, CCFull iterates over the hyperedges and checks if data mining queries connected with the current hyperedge have been already assigned to phases (partitions). If none of the data mining queries has been assigned so far, and if their hash trees fit in memory, then a new phase is generated and the data mining queries are assigned to it. Otherwise, if only some of the data mining queries have been already assigned to different phases, then CCFull tries to combine all those phases together with the unassigned data mining queries. If such combined phase does not fit in memory, then the current hyperedge is ignored and CCFull continues with the next one. The algorithm ends when all hyperedges are processed. The algorithm CCFull is shown in Fig. 9.

```
CCFull(G=(V,E)):
 begin
  $Phases = \{\emptyset\}$
  sort $E = <e_1, e_2,..., e_k>$ in desc. order w.r.t. $e_i$.gain, ignore edges with zero gains
  for each $e_i$ in $E$ do begin
    $tmpV = \{v \in V \mid v \in e_i\}$
    if $(|\{p \in Phases \mid p \cap tmpV \neq \emptyset\}| = 0)$ then
      $commonPhases = \emptyset$
      $newPhase = tmpV$
 else
      $commonPhases = \{p \in Phases \mid p \cap tmpV \neq \emptyset\}$
      $newPhase = tmpV \cup \bigcup p \mid p \in commonPhases$
 end if
    if $(treesize(newPhase) \leq MEMSIZE)$ then
      $Phases = Phases - commonPhases$
      $Phases = Phases \cup newPhase$
    end if
  end
  add phase for each unassigned query
  return Phases
 end
```

Fig. 9. CCFull

The detailed steps of the CCFull algorithm are the following. First we initialize the set of phases – we start with the empty set. In the next step we sort the list *E* of hyperedges from the gain graph. Hyperedges with weights equal to zero are removed from the list. Then a loop starts, which iterates over the list of hyperedges. In the first step of the loop we select all data mining queries which are connected with the current hyperedge (*tmpV*). Next we test if any of the selected data mining queries belongs to any of the phases created so far. If not, then we create a new candidate phase containing all the data mining queries from *tmpV*. Otherwise, we create a new candidate phase containing all the data mining queries from *tmpV* and data mining queries from earlier created phases, to which any of the *tmpV* data mining queries was also assigned. After the process of building the new candidate phase is completed, we check if hash trees of all the data mining queries from it fit together in memory (*MEMSIZE* is the available memory size). If this condition is satisfied, then we append the new candidate phase to the current set of created phases *Phases*, possibly replacing some of the existing phases (when multiple phases are combined). Finally, when the loop is finished, for each data mining query which has not been assigned we create a new phase.

7.3 CCCoarsening

Earlier we have stated that hypergraph partitioning algorithms from other domains are not applicable to our problem due to its specifics. Nevertheless, the existing hypergraph partitioning algorithms can provide inspiration for the development of new methods. This is exactly the case with CCCoarsening which borrows ideas from the heavy edge matching method of graph coarsening in multi-level graph partitioning [33].

The CCCoarsening algorithm starts with transformation of the data sharing hypergraph into a *gain graph*, which contains (1) vertices being the original data mining queries and (2) two-vertex edges whose weights describe gains that can be reached by executing the connected queries in the same phase. The idea is to avoid the problem of an exponential number of hyperedges (with respect to the number of vertices) suffered by CCFull, which uses a gain hypergraph. The price for the above simplification is the loss of precise information on actual gains due to assigning any given subset of queries to the same phase. As a consequence, only pairs of queries/phases will be considered for merging in each step of the iterative process.

A sample gain graph for the set of three data mining queries is shown in Fig. 10. For example, putting the data mining queries dmq_1 and dmq_2 in the same phase would allow us to save 90 I/O cost units (e.g., disk blocks).

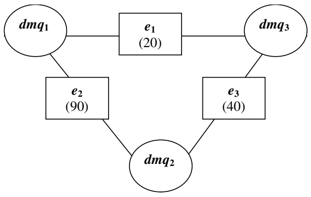

Fig. 10. Sample gain graph

The heavy edge matching method [33], of which CCCoarsening is an adaptation, was designed as a method of coarsening large graphs by collapsing strongly connected vertices. The original heavy-edge matching algorithm iteratively reduces the graph by minimizing the number of its vertices. In each iteration, the algorithm looks for the maximal set of edges (called *matching*) such that it contains no pair of edges incident with the same vertex. In order to generate a matching, a vertex currently not matched is randomly selected. Then, from the set of edges incident to the selected vertex an edge of the maximal weight from the edges leading to other so far unmatched vertices is chosen. The chosen edge results in selection of its two incident vertices for matching and labeling them as „matched”. When the matching is complete, the edges forming it are removed from the graph and each pair of vertices connected by an edge that is to be removed are merged into one vertex, whose weight is the sum of weights of the two merged vertices. After merging vertices, some of the remaining edges may also merge as only single edge between any pair of vertices is allowed. If a set of edges is being merged into one edge, its resulting weight is the sum of the weights of the replaced edges.

Our adaptation of the heavy-edge matching algorithm for the purpose of query set partitioning for Common Counting focuses on the modification of the vertex reduction step. When an unmatched vertex is randomly selected, we will sort the edges connecting it with other unmatched vertices in the decreasing order of weights. Next, we choose the edge with the highest weight and check if the weight of a new vertex that would be created by collapsing the selected edge does not exceed the limit on the vertex weight (representing the amount of available memory to store hash trees). If not, the pair of vertices connected by the selected edge is merged. Otherwise, the edge next in order is chosen from the sorted list of edges incident to the randomly selected vertex. If none of edges incident to the vertex leads to a feasible merging, the vertex is not merged but still it is marked as matched so it will not be considered again in the current iteration of the graph coarsening process. The algorithm finishes work if no feasible matching of vertices can be found in a new coarsening iteration. The vertices of the resulting reduced graph represent partitions corresponding to phases of our original problem.

7.4 CCAgglomerative

Similarly to CCCoarsening, the CCAgglomerative algorithm first transforms the data sharing graph into a gain graph, but then uses a different method of grouping queries into phases. In CCCoarsening a matched vertex is not considered for subsequent merges until the matching in the present coarsening iteration is completed. CCAgglomerative does not pose such a restriction.

```
CCAgglomerative($G$=($V$,$E$), $E$ contains 2-node edges only):
begin
  $Phases = \emptyset$
  for each $v$ in $V$ do $Phases = Phases \cup \{\{v\}\}$
  sort $E = \{e_1, e_2, ..., e_k\}$ in descending order w.r.t. $e_i$.gain, ignore edges with zero gains
  for each $e_i = (v_1, v_2)$ in $E$ do begin
     $phase_1 = p \in Phases$ such that $v_1 \in p$
     $phase_2 = p \in Phases$ such that $v_2 \in p$
     if $treesize(phase_1 \cup phase_2) \leq MEMSIZE$ then
       $Phases = Phases - \{phase_1\}$
       $Phases = Phases - \{phase_2\}$
       $Phases = Phases \cup \{phase_1 \cup phase_2\}$
     end if
  end
  return Phases
end
```

Fig. 11. CCAgglomerative

CCAgglomerative starts with an initial partitioning created by putting each data mining query into a separate phase. Next, the algorithm processes the edges sorted with respect to the decreasing weights. For each edge, the algorithm tries to combine phases containing the connected data mining queries into one phase. If the total size of all the data mining queries in such phase does not exceed the memory size, the original phases are replaced with the new one. Otherwise the algorithm simply ignores the edge and continues. The CCAgglomerative algorithm is shown on Fig. 11.

7.5 CCAgglomerativeNoise

Algorithm CCAgglomerative is a heuristics that suffers from the same problem as classical greedy algorithms. (We will present a greedy algorithm for our problem in Sect. 7.6 and then improve it in Sect. 7.7.) Merging phases connected by the heaviest edge in each iteration may not always lead to the optimal assignment of queries to phases. Let us consider an example gain graph representing a batch of queries shown in Fig. 12.

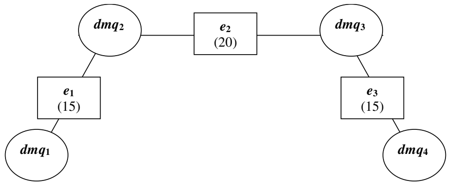

Fig. 12. Example gain graph for which CCAgglomerative misses the optimal solution

Assume that in a certain iteration of Common Counting the sizes of candidate hash-trees are 20 KB for all four queries, and the amount of available memory is 40KB, which means that no more than two queries can be processed in one phase. In such a case, CCAgglomerative would start with assigning dmq_2 and dmq_3 to the same phase, and then dmq_1 and dmq_4 would be assigned to different phases. The reduction in number of disk blocks read, compared to sequential execution, would be 20 blocks. Obviously, the optimal solution is to execute dmq_1 and dmq_2 in one phase and dmq_3 and dmq_4 in another, leading to the gain of 30 blocks.

To give the partitioning algorithm a chance of finding an optimal assignment, we propose to randomize the graph by randomly modifying weights of graph edges within a user-specified window (expressed in percents, e.g., ±10%), and then execute the unmodified CCAgglomerative algorithm on a modified gain graph[7]. The procedure of randomizing the graph and partitioning should be repeated a user-specified number of times, each time starting with the original gain graph. We call the extended partitioning algorithm CCAgglomerativeNoise as it introduces some "noise" into the graph model of the batch of queries, before performing actual partitioning. For the noise of *x%*, in a randomized gain graph the modified weight *e.gain'* of each edge *e* will be a random number from the range <*e.gain-x%*e.gain*, *e.gain+x%*e.gain*>, where *e.gain* is the original weight of the edge *e*.

[7] Iterative execution of a partitioning heuristics over a randomized data sharing model could also be considered for the partitioning algorithms presented in previous sections. We implemented this idea in the context of CCAgglomerative as at the time it was the most efficient of the algorithms proposed so far.

To illustrate a potential usefulness of CCAgglomerativeNoise let us go back to the example gain graph from Fig. 12. For the noise of 20%, in each iteration of CCAgglomerativeNoise modified values of edge weights would be from the following ranges: $e_1.gain' \in$ <12,18>, $e_2.gain' \in$ <16,24>, and $e_3.gain' \in$ <12,18>. So, it is possible that in some iteration of CCAgglomerativeNoise we would have $e_1.gain' > e_2.gain'$ or $e_3.gain' > e_2.gain'$ (e.g., $e_1.gain' = 18$, $e_2.gain' = 16$, and $e_3.gain' = 13$), in which case the basic CCAgglomerative partitioning procedure would find the optimal assignment of queries to Common Counting phases.

We should note that the CCAgglomerativeNoise method should be treated as a means of improving the results of pure CCAgglomerative. In other words, the initial iteration of CCAgglomerativeNoise should always be on the original gain graph. This way it can be guaranteed that CCAgglomerativeNoise will never generate worse partitionings than CCAgglomerative.

7.6 CCGreedy

The general greedy strategy can be applied to solve the hypergraph partitioning problem representing query set partitioning for Common Counting by starting with each query in a separate partition and then iteratively merging pairs of partitions, greedily choosing the two partitions whose merging results in greater improvement of the partitioning objective and at the same time does not violate the partitioning constraint. This leads to the CCGreedy algorithm presented in Fig. 13.

```
CCGreedy(GG=(V,E)):
begin
  while (true) begin
    sort E in descending order w.r.t. $e_i$.gain, ignore edges with zero gains
    newPartition = $\emptyset$
    for each $e_i = \{v_x, v_y\}$ in E do
      if (treesize($e_i$) $\leq$ MEMSIZE) then
        newPartition = $v_x \cup v_y$
        $V = V \setminus \{v_x, v_y\}$
        $V = V \cup \{newPartition\}$
        $E = E \setminus e_i$
        for each v in V do begin
          newEdge = {v, newPartition}; compute newEdge.gain
          $E = E \cup \{newEdge\}$
        end
        break
      end if
    end
    if newPartition = $\emptyset$ then break end if
  end
  return V
end
```

Fig. 13. CCGreedy

To represent the gain in the partitioning objective for all pairs of partitions the algorithm maintains a *gain graph GG*=(*V*, *E*), which is a fully connected graph whose nodes represent partitions and each edge weight represents the gain thanks to merging a pair of partitions connected by the edge. The gain is computed as the difference between the values of partitioning objectives after and before merging a given pair of queries.

It should be noted that while CCCoarsening and CCAgglomerative also use the same gain graph structure as CCGreedy, the advantage of CCGreedy is that it updates the gain graph after merging partitions so that the graph always reflects possible gains due to partition merging.

7.7 CCSemiGreedy

An obvious problem with greedy algorithms like CCGreedy is that the locally optimal choice in each operation may not lead to the globally optimal solution. To increase the chances of finding the optimal partitioning we modify CCGreedy by applying a semi-greedy strategy [21] to it. The result is the CCSemiGreedy algorithm depicted in Fig.14.

```
CCSemiGreedy(GG=(V,E), RCLLen):
begin
 while (true) begin
   sort E in desc. order w.r.t. $e_i$.gain,
   ignore edges with zero gains
   newPartition = $\emptyset$
  RCL = genRCL(GG, RCLLen)
  if length(RCL) = 0 then break end if
  randomly choose $e_i = \{v_x, v_y\}$ from RCL
  newPartition = $v_x \cup v_y$
  $V = V \setminus \{v_x, v_y\}$
  $V = V \cup \{newPartition\}$
  $E = E \setminus e_i$
  for each v in V do begin
   newEdge = {v, newPartition}
   compute newEdge.gain
   $E = E \cup \{newEdge\}$
  end
 end
 return V
end
```

```
function genRCL(GG=(V,E), RCLLen):
begin
 RCL = nil
 for each $e_i = \{v_x, v_y\}$ in E do
  if ($treesize(e_i) \leq MEMSIZE$) then
   RCL = append(RCL, $e_i$)
   if length(RCL) = RCLLen then
    break
   end if
  end if
 end
 return RCL
end
```

Fig. 14. CCSemiGreedy

CCSemiGreedy differs from CCGreedy in the step of choosing the partitions to merge. CCSemiGreedy uses restricted candidate list (*RCL*) which is returned by the function *genRCL*. This procedure iterates over the gain graph and checks if hash trees of all the queries from a given pair of partitions fit together in memory. If this condition is satisfied, the current edge is added to the *RCL*. Generation of the *RCL* is stopped when the list reaches the length of *RCLLen* (set by a user). In CCSemiGreedy we check the length of the *RCL*. If it is zero, there is no possible merge, otherwise an edge (for partition merging) is chosen randomly from the *RCL*. Other steps of the CCSemiGreedy algorithm are the same as those described for CCGreedy algorithm.

In practice, CCSemiGreedy should be applied to query partitioning in the following way. Firstly, an initial partitioning should be generated with CCGreedy. Then, CCSemiGreedy should be executed a user-defined number of times. In the end, the best of the generated partitionings should be used for Common Counting.

8 Experimental Results

This section contains the results of experiments that we conducted to compare the proposed query partitioning algorithms for Common Counting. Due to the large number of the algorithms to compare, the experiments were divided into a couple of stages. The first stage was devoted to comparison of basic versions of the partitioning algorithms dedicated to our particular problem and developed with the goal of solving it in mind, i.e. CCRecursive, CCFull, CCCoarsening, and CCAgglomerative. The second stage contained experiments aimed at comparing the best algorithm selected in the first stage, which turned out to be CCAgglomerative, with CCGreedy and CCSemiGreedy, which are adaptations of the universal greedy and semi-greedy metaheuristics. Also at that stage CCAgglomerativeNoise, an extension of CCAgglomerative, somewhat analogous to CCSemiGreedy with respect to CCGreedy, was included in the tests.

In both stages, the experiments were performed on a synthetic dataset generated with GEN [2]. The dataset had the following characteristics: number of transactions = 500000, average number of items in a transaction = 4, number of different items = 10000, number of patterns = 1000. The size of the dataset in a textual form was about 16MB. We stored the dataset in a local PostgreSQL database, where it consumed about 85MB of disk space + extra 43 MB used for a B-tree index.

Batches of frequent itemset were generated with our own random generator parameterized with a desired average percentage of dataset overlapping between pairs of queries from a batch. The minimum support (frequency) threshold of all the queries was always set to 0.75%.

For the above support threshold the hash trees usually had the size of tens of kilobytes. In order to introduce the need for query set partitioning, we deliberately limited the amount of memory available for Common Counting executions to 120kB (different values were also used in one of the experiments)[8]. Obviously, typically for sparse datasets, as those generated with GEN, the hash trees were biggest in the second and third Apriori iteration, and getting smaller with each subsequent iteration. As a consequence, the need for query set partitioning was observed only for some of the Common Counting iterations, which is typical for real-life scenarios with datasets whose characteristics the GEN generator tries to mimic.

[8] Alternatively, we could have changed the parameters of the GEN generator and/or decrease the minimum support threshold of the queries. The advantage of our approach was that it kept the execution times reasonable without the loss of generality.

8.1 Comparison of Basic Dedicated Algorithms

In this stage of experiments we tested the four dedicated algorithms: CCRecursive, CCFull, CCCoarsening, and CCAgglomerative, comparing them to two extra algorithms provided as reference points: an exact algorithm (denoted as "Exact" in the charts) enumerating all feasible partitionings by performing a brute-force search and an algorithm randomly assigning queries to partitions so that the partitioning constraint is satisfied denoted as "Random" in the charts). These two extra algorithms were ultimate choices to provide the reference points for judging usefulness of our proposed algorithms as:

- a heuristic algorithm should generate solutions as close to those returned by an exact algorithm,
- an algorithm generating solutions worse than generated randomly is obviously useless.

During the experiments we measured total execution time, time consumed by a partitioning algorithm, the number of partitions in partitionings, and the number of disk blocks read in a Common Counting iteration for a generated query set partitioning. The experiments were conducted on a PC with AMD Athlon 1400+ processor and 384 MB of RAM running Windows XP. The data resided in a local PostgreSQL database, the algorithms were implemented in C#.

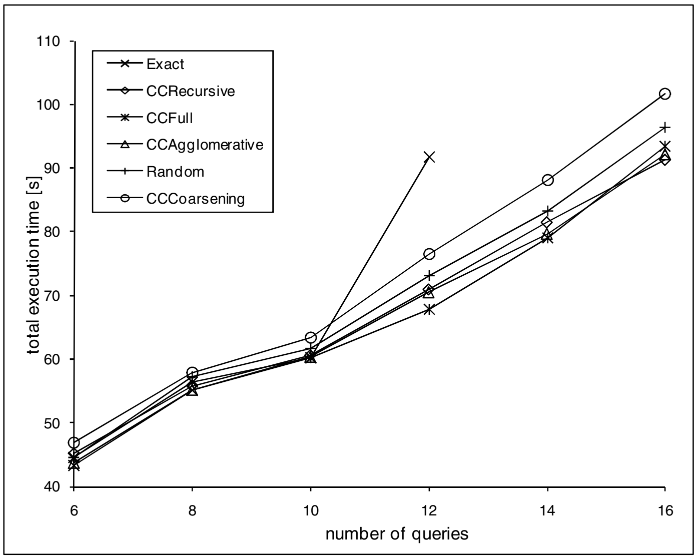

Fig. 15. Total execution time of a batch of queries (40% overlapping, 120kB of memory)

Figure 15 presents total execution times of a batch of randomly generated queries using the Common Counting method equipped with different query set partitioning algorithms. The queries were generated so that the average overlapping between their datasets was 40%. The memory available was limited to 120kB. The number of queries varied from 6 to 16. The exact algorithm finished in reasonable time only for up to 12 queries (it did not complete in 900s for the case of 14 queries), which is a practical confirmation of our theoretical analysis suggesting that in order to support large batches of queries heuristic partitioning algorithms are required. The two best algorithms in terms of overall processing time of Common Counting are CCAgglomerative and CCFull, with CCRecursive not far behind. However, while CCAgglomerative has polynomial computational complexity with respect to the number of queries, the complexity of CCFull is exponential. This is due to the fact that CCAgglomerative iterates over edges in a connected graph, while CCFull does the same for hyperedges in a hypergraph. The result is observed deterioration of execution times with the increasing number of queries when CCFull was applied, compared to the characteristic of CCAgglomerative. The worst of the four proposed algorithms is CCCoarsening, which performed worse than the random approach.

Obviously, the total execution times are what matters in the end. Nevertheless, to provide an insight into reasons of the overall performance of the tested algorithms, we also analyzed separately the two factors that contributed to differences in the total execution times: the time consumed by partitioning algorithms themselves and the quality of generated partitionings, measured as the number of disk blocks read in a Common Counting iteration due to a generated partitioning.

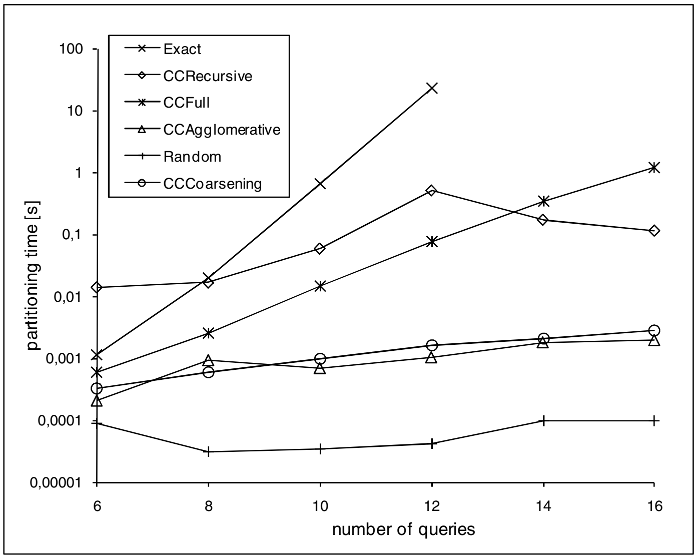

Fig. 16. Partitioning execution time (40% overlapping, 120kB of memory)

The execution times of partitioning algorithms measured in the experiment are depicted in Fig. 16. Due to large disproportions between the algorithms, the chart uses logarithmic scale. The chart confirms that the execution times of the exact algorithm and CCFull grow exponentially with the number of queries, which prohibits application of both to large batches of queries. Still, as far as the partitioning time in a concern, CCFull is an improvement over the exact algorithm as it completed within a second even for 16 queries, which is acceptable as almost negligible compared to the overall execution time of Common Counting. The fastest algorithms (apart from the random approach) are CCAgglomerative, and CCCoarsening. CCRecursive is the least predictive of the algorithms and for small batches of queries is even slower than CCFull.

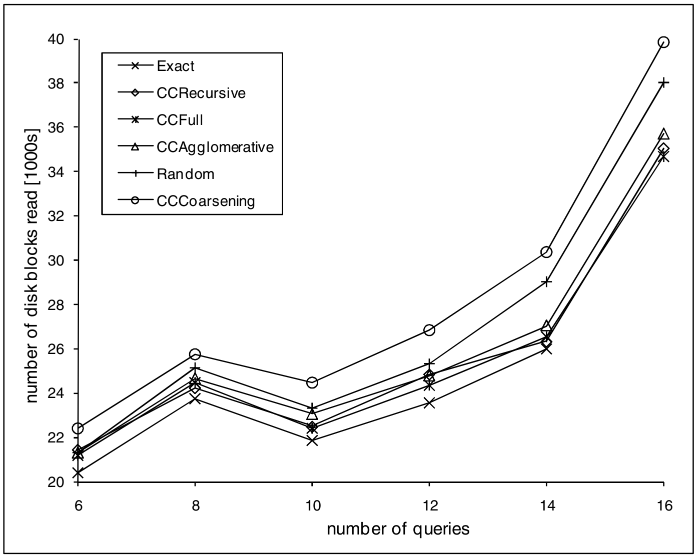

Fig. 17. Number of disk blocks read for a generated partitioning (40% overlapping, 120kB of memory)

Figure 17 shows the numbers of disk blocks read in a Common Counting iteration due to a partitioning generated with a given algorithm, which reflects the accuracy of the partitioning algorithms. The results are generally consistent with the total execution times of Common Counting as the time spend on partitioning was a small fraction of the time spent on database scans. CCFull, CCAgglomerative, and CCRecursive generate partitioning of similar quality, within 5% of those generated by the exact algorithm, while CCCoarsening is worse than the random algorithm. As for the influence of the number of queries on the quality of partitioning, the most important observation is that the relative accuracy of CCFull, CCAgglomerative, CCRecursive, and CCCoarsening with respect to exact and random algorithms was roughly the same for all sizes of query batches. The absolute number of disk blocks read does not necessarily rise with the increase of the number of queries as the complexity of the data sharing hypergraph does not have to be greater for a larger number of randomly generated queries. Nevertheless, still such a tendency can be observed despite the aberration for the case of 8 queries.

Figure 18 provides an explanation of poor accuracy of CCCoarsening. The average number of partitions in partitionings generated by the tested algorithms is presented. Evidently, CCCoarsening is worse than the rest of the algorithms by 0.5 to 1 iteration on average. The reason for this is the origin of the concept underlying the CCCoarsening algorithm. When the actual goal is coarsening the graph, having partitions of similar size is desired to preserve the general structure of a graph. However, in the case of our problem, the fact that the algorithm was focusing on growing several partitions at the same pace reduces the possibility of fully exploiting the memory available as merging partitions into larger ones becomes impossible quicker than in case of other algorithms. Thus, the possibilities of minimizing our partitioning criterion are reduced in the case of CCCoarsening for the sake of balancing the sizes of partitions, which is irrelevant for our problem.

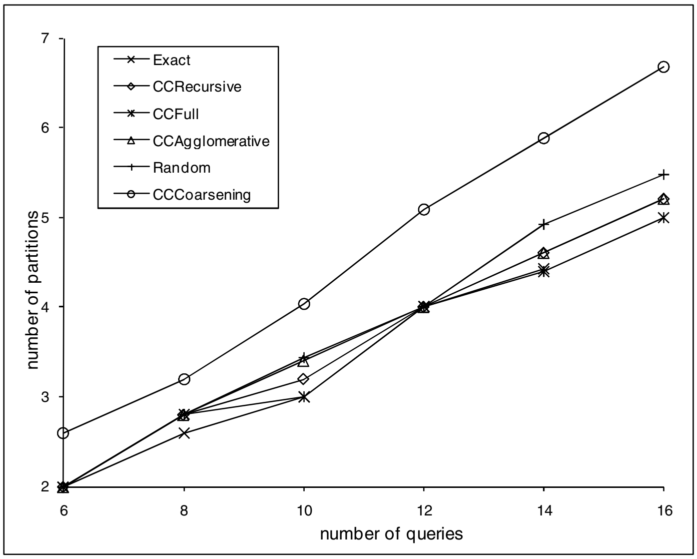

Fig. 18. Average number of partitions in a partitioning (40% overlapping, 120kB of memory)

The last goal of this stage of experiments was testing the impact of the level of dataset overlapping between the queries within a batch on the performance of Common Counting with various partitioning algorithms. Figures 19 and 20 show the total execution times for four different average levels of query dataset overlapping with 10 and 40 queries in a batch respectively (for 40 queries the exact algorithm and CCFull could not complete within reasonable time and therefore are not included in the results).

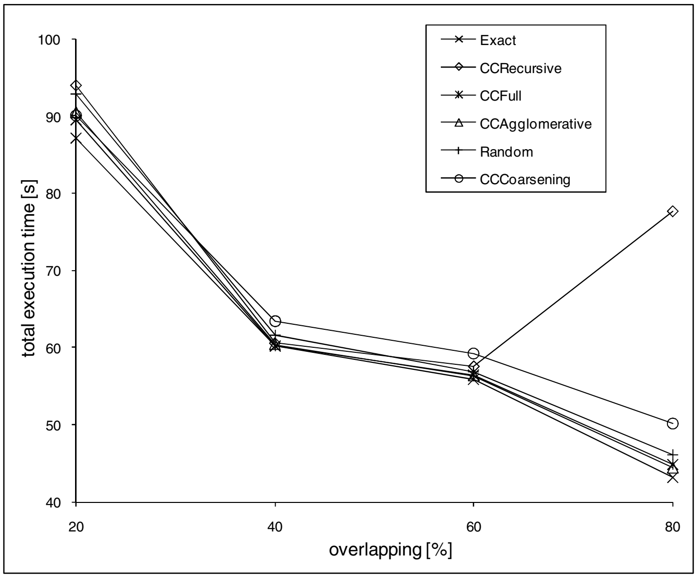

Fig. 19. Total execution time of a batch of queries (10 queries, 120kB of memory)

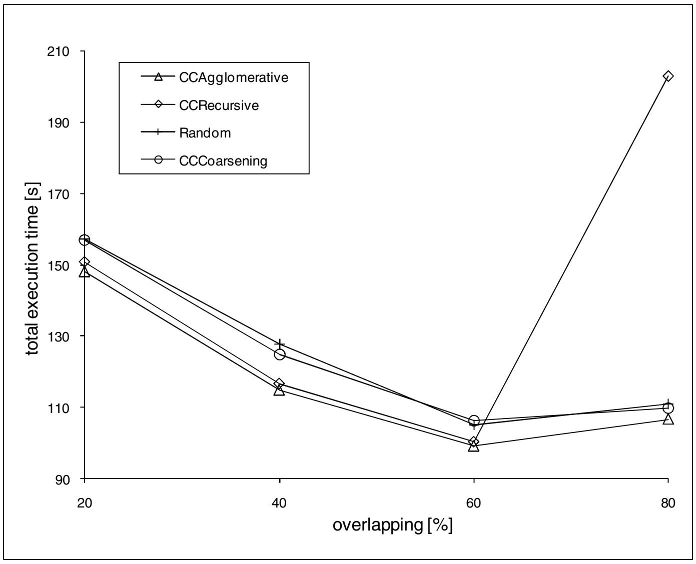

Fig. 20. Total execution time of a batch of queries (40 queries, 120kB of memory)

In general, the results are consistent with expectations as the more significant the overlapping the better chances of reducing the I/O cost of database scanning. A negative surprise is a significant performance degradation of CCRecursive for the average overlapping of 80% (the rest of algorithms exhibit slight performance degradation only for the case of 40 queries as compared with the overlapping of 60%). We believe that such a behavior can be explained by the recursive nature of CCRecursive. When the queries almost completely overlap, there is a large number of elementary data selection predicates corresponding to small dataset partitions, share by a large number of queries. As a result, CCRecursive violates the partitioning constraint while there are still many elementary data selection predicates to process (recall that CCRecursive is the only of the algorithms working with the predicates one by one). This leads to a lot of recursive calls due to attempts of creating partitions exceeding the size limit and consequently deteriorates CCRecursive's performance.

The overall conclusion from this stage of experiments is that the best out of four proposed algorithms dedicated to our query set partitioning problem is CCAgglomerative as one of the two algorithms tied for the first place both in terms of partitioning time and quality.

8.2 Comparison of Greedy Approaches with the Best Dedicated Algorithms

The goal of the second stage of experiments was comparison of the best of dedicated partitioning algorithms, which turned out to be CCAgglomerative, with an implementation of a greedy strategy – CCGreedy. Also included in the tests and evaluated were their extensions, namely: CCAgglomerativeNoise and CCSemiGreedy. Before the actual tests of the partitioning speed and accuracy of the compared algorithms, the optimal values of parameters responsible for the chance of improving the initial solution of the two basic compared algorithms had to be determined. All the tested algorithms were implemented C#. The experiments were conducted on a PC with Intel Pentium IV 2.53GHz processor and 512MB of RAM running Windows XP.

We started the experiments with simulations, performed to determine influence of CCSemiGreedy parameters (RCL length and number of attempts) on its effectiveness. We simulated batches of data mining queries by randomly generating the database predicate and size of the candidate tree for each query. Size of available memory was randomly generated in such way that at least every single query could fit into memory. Series of simulations consisted of 500 iterations to get average values and were applied to batches of queries ranging from 3 to 50 queries per batch.

Figure 21 presents the influence of chosen RCL length on the number of disk blocks read by CCSemiGreedy. The experiments indicate that the length of the RCL should be very small but greater than 2 items. Best results were obtained for 3 to 6 items. For further experiments we have chosen the length of RCL equal to 3.

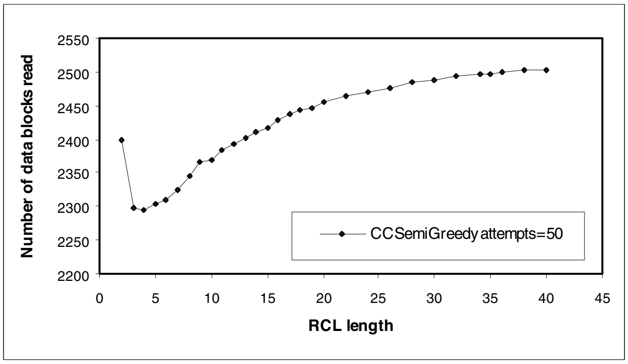

Fig. 21. Influence of the RCL length on the overall accuracy of CCSemiGreedy

Figure 22 presents influence of the second parameter of CCSemiGreedy, which is the number of attempts to generate a partitioning. It is obvious that more attempts generally will result in better partitionings but at the expense of increasing the partitioning time. Results indicate that after more than fifty attempts there is no significant improvement in the quality of the partitioning.

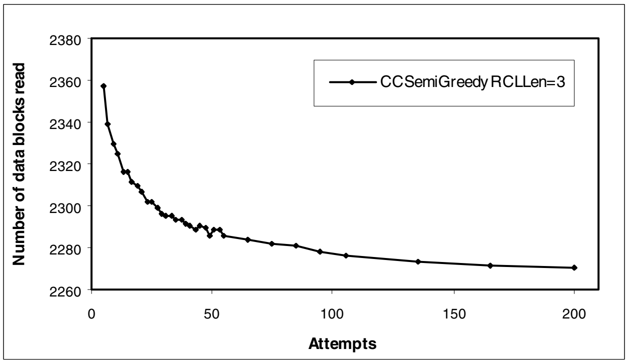

Fig. 22. Influence of the number of attempts on the overall accuracy of CCSemiGreedy

CCAgglomerativeNoise iteratively tries to improve the partitioning generated by CCAgglomerative in a similar way as CCSemiGreedy extends CCGreedy and is also parameterized by the number of iterations. We set the number of attempts to 150 for CCAgglomerativeNoise because this value resulted in CCSemiGreedy and CCAgglomerativeNoise consuming roughly equal time to generate the partitioning for the average size of batches used in the planned experiments. For that number of iterations we determined the optimal value of the noise parameter of CCAgglomerativeNoise in similar simulations to those carried for RCL length of CCSemiGreedy. The influence of the noise parameter on CCAgglomerativeNoise was analogous to that of RCL length on CCSemiGreedy. The optimal value of noise turned out to be 3%, which is relatively small.

Knowing the optimal values of parameters of CCSemiGreedy and CCAgglomerativeNoise, we used these values in the subsequent experiments in which we compared CCGreedy and CCSemiGreedy algorithms with CCAgglomerative and CCAgglomerativeNoise in terms of effectiveness (quality of generated partitionings) and efficiency (partitioning times). In these experiments we randomly generated batches of 5 to 30 queries, operating on subsets of the test database.

Figure 23 presents how the accuracy of the partitioning algorithms changes with the number of queries. To improve readability of the chart, we present relative amount of data blocks read as result of partitionings generated by CCGreedy, CCSemiGreedy and CCAgglomerativeNoise with respect to CCAgglomerative[9]. Experiments were performed for three values of the main memory limit (90, 120, and 150kB) and for four levels of the average overlapping of datasets read by queries in the set (20%, 40%, 60%, and 80%). The results presented are averages taken over all the conducted experiments. Results show that the most effective partitionings are generated by CCSemiGreedy and are about 5% better than those generated by CCAgglomerative. For CCAgglomerativeNoise and CCGreedy the measured improvement over CCAgglomerative was 2% and 1% respectively.

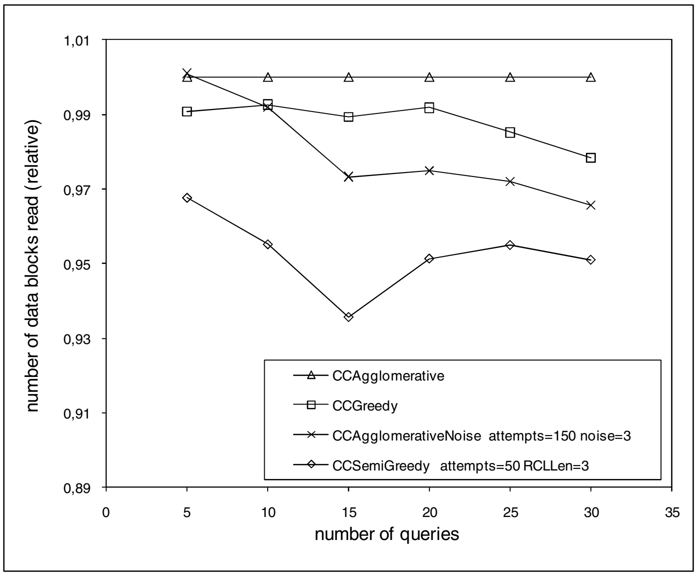

Fig. 23. Amounts of data read by different partitionings

[9] For this stage of experiments we do not present the total execution times of Common Counting because the difference between the algorithms would be difficult to notice from the chart since the difference in the execution times among the algorithms was two orders of magnitude smaller than the difference between execution times for the smallest and largest of query batches.

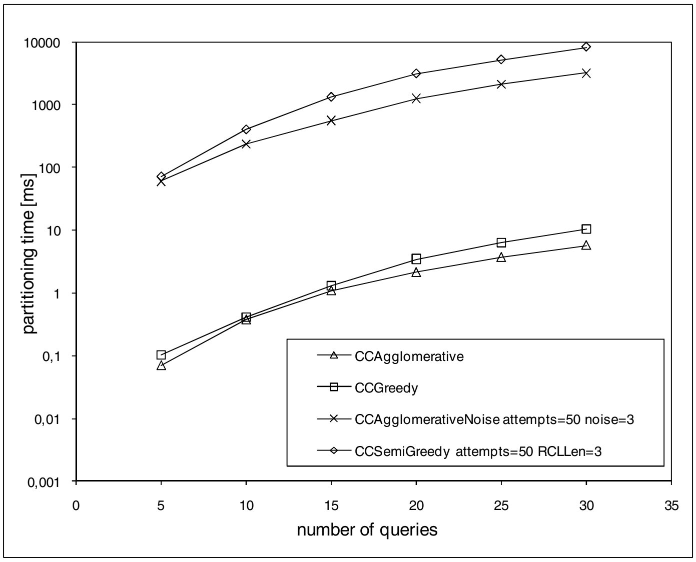

Fig. 24. Partitioning times (logarithmic scale)

Figure 24 presents execution times of the considered partitioning algorithms. This time for CCSemiGreedy and CCAgglomerativeNoise numbers of attempts were fixed at the same level (50). Execution times of CCAgglomerative and CCGreedy are negligible, with CCGreedy requiring at most twice as much time as CCAgglomerative. Execution times of CCSemiGreedy are up to three times longer than those of CCAgglomerativeNoise and the gap increases with the number of queries.

The results of our experiments show that CCGreedy is more effective than CCAgglomerative, and properly parameterized CCSemiGreedy generates better partitionings than CCAgglomerativeNoise, which makes it the best partitioning algorithm for Common Counting. The execution times of the new algorithms are longer but in typical situations the increase in partitioning time will be dominated by the reduction of the time spent on disk operations thanks to better partitionings.

9 Review of Other Methods of Processing Sets of Frequent Itemset Queries

While Common Counting is the most fundamental, and at the same time predictable in terms of offered performance gains with respect to sequential execution, method of processing sets of frequent itemset queries, it is not the only possible solution of the considered problem. Two general approaches have been taken to design methods of processing batches of frequent itemset queries: (1) providing methods independent from a particular frequent pattern mining algorithm, and (2) tailoring dedicated methods for the two most prominent frequent pattern mining algorithms, i.e., Apriori and FP-growth. Obviously, Common Counting is a representative of the second approach.

The first method independent of the mining algorithm was Mine Merge [55] which transformed the original batch of queries into a set of intermediate queries operating on non-overlapping parts of the database. The results of these intermediate queries were used to answer the original queries during a verifying pass over the database. Mine Merge was shown to scale poorly with the number of queries due to exponential growth of the number of resulting intermediate queries. Moreover, it requires significant overlapping among the queries' datasets in order to compensate the extra database pass. The latter problem has been solved by a modified version of Mine Merge, called Partition Mine Merge Improved [17], which generated intermediate queries whose source datasets could be cached in memory and thus required exactly two scans of the database to process the batch of queries. The price for the reduction of I/O was the increase in the number of intermediate queries, resulting in the increased amount of in-memory computations.

Following Common Counting, two methods offering tighter integration of processing among the concurrently executed queries were proposed for Apriori. Common Candidate Tree [16] replaced individual hash trees with one integrated data structure shared by all the queries, thus reducing the memory consumption and optimizing the candidate counting step of Apriori. Later, Common Candidates [29] integrated also the candidate generation step of Apriori, while preserving all the optimizations proposed earlier in Common Counting and Common Candidate Tree. Unfortunately, the successors of Common Counting do not preserve its capability of handling large batches of queries by partitioning them into phases, thus being restricted by the limit of memory available for the integrated in-memory data structure. Furthermore, Common Candidates limits the possibilities of constraint handling due to replacing the original candidate generation procedure of Apriori which serves as the basis for most of the constraint-handling techniques within the Apriori framework.

As for FP-growth, the methods of processing sets of frequent itemset queries using this algorithm evolved analogously to the ones for Apriori [50]. The initial proposal was Common Building, a direct adaptation of Common Counting, which integrated database scans performed by the queries in order to build their FP-tree structures in main memory. The method was immediately extended by introducing a variation of FP-tree that could be shared by the batch of queries, resulting in the Common FP-tree method.

10 Conclusions

In this chapter we considered the problem of processing sets of frequent itemset queries, which brings the ideas of multiple-query optimization to the domain of data mining as a natural consequence of previous research on optimizing individual frequent itemset queries and sequences of frequent itemset queries.

We provided a general model of frequent itemset queries, which was then used as a basis for the formulation of our multi-query optimization problem. From the algorithms dedicated to solve the problem that we had proposed over the last decade, here we focused on the most fundamental and predictable method, called Common Counting, which consists in concurrent execution of the queries using Apriori with the integration of scans of the parts of the database shared among the queries.

The major advantage of Common Counting over its alternatives is its applicability to arbitrarily large batches of queries. In order to achieve that feature, Common Counting had to be accompanied with a method of dealing with situations when hash trees of all the queries do not fit together in main memory. The problem of limited memory was addressed by partitioning the set of queries into subsets processed in several phases. This approach led to an interesting optimization problem that we formalized as a specific case of hypergraph partitioning. Since the problem is NP-hard, it has to be solved by heuristic algorithms in case of large batches of queries.

For the identified, specific hypergraph partitioning problem, we provided a comprehensive overview of query set partitioning algorithms proposed by us so far: CCRecursive, CCFull, CCCoarsening, CCAgglomerative, CCAgglomerativeNoise, CCGreedy, and CCSemiGreedy.

Finally, we presented extensive results of experiments aimed at evaluating the performance and accuracy of the algorithms. The results indicate that the implementation of the universal greedy metaheuristics (CCGreedy) and its semi-greedy extension (CCSemiGreedy) generate better partitionings in terms of resulting I/O costs of Common Counting than those generated by the best algorithms dedicated to our query set partitioning problem, while offering satisfactory (but not shortest) execution times.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Mining Association Rules Between Sets of Items in Large Databases. In: Buneman, P., Jajodia, S. (eds.) Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, pp. 207–216. ACM Press, New York (1993)
2. Agrawal, R., Mehta, M., Shafer, J., Srikant, R., Arning, A., Bollinger, T.: The Quest Data Mining System. In: Simoudis, E., Han, J., Fayyad, U. (eds.) Proceedings of the Second International Conference on Knowledge Discovery in Databases and Data Mining, pp. 244–249. AAAI Press, Menlo Park (1996)
3. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487–499. Morgan Kaufmann, San Francisco (1994)
4. Alpert, C.J., Kahng, A.B.: Recent Directions in Netlist Partitioning: A Survey. Integration: The VLSI Journal 19, 1–81 (1995)
5. Alsabbagh, J.R., Raghavan, V.V.: Analysis of common subexpression exploitation models in multiple-query processing. In: Rusinkiewicz, M. (ed.) Proceedings of the 10th International Conference on Data Engineering, pp. 488–497. IEEE Computer Society, Los Alamitos (1994)
6. Baralis, E., Psaila, G.: Incremental Refinement of Mining Queries. In: Mohania, M., Tjoa, A.M. (eds.) DaWaK 1999. LNCS, vol. 1676, pp. 173–182. Springer, Heidelberg (1999)
7. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research 16, 135–166 (2002)
8. Boinski, P., Jozwiak, K., Wojciechowski, M., Zakrzewicz, M.: Improving Quality of Agglomerative Scheduling in Concurrent Processing of Frequent Itemset Queries. In: Klopotek, M.A., Wierzchon, S.T., Trojanowski, K. (eds.) Proceedings of the International IIS: IIPWM 2006 Conference, pp. 233–242. Springer, Heidelberg (2006)

9. Boinski, P., Jozwiak, K., Wojciechowski, M., Zakrzewicz, M.: Estimating Hash-Tree Sizes in Concurrent Processing of Frequent Itemset Queries. International Journal of Information Technology and Intelligent Computing 1, 405–417 (2006)
10. Boinski, P., Wojciechowski, M., Zakrzewicz, M.: A Greedy Approach to Concurrent Processing of Frequent Itemset Queries. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2006. LNCS, vol. 4081, pp. 292–301. Springer, Heidelberg (2006)
11. Ceri, S., Meo, R., Psaila, G.: A New SQL-like Operator for Mining Association Rules. In: Vijayaraman, T.M., Buchmann, A.P., Mohan, C., Sarda, N.L. (eds.) Proceedings of the 22th International Conference on Very Large Data Bases, pp. 122–133. Morgan Kaufmann, San Francisco (1996)
12. Cheung, D.W., Han, J., Ng, V.T., Wong, C.Y.: Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. In: Su, S.Y.W. (ed.) Proceedings of the 12th International Conference on Data Engineering, pp. 106–114. IEEE Computer Society, Los Alamitos (1996)
13. Cheung, D.W., Lee, S.D., Kao, B.: A General Incremental Technique for Maintaining Discovered Association Rules. In: Topor, R.W., Tanaka, K. (eds.) Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, pp. 185–194. World Scientific, Singapore (1997)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability. A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, San Francisco (1979)
15. Goethals, B., Van den Bussche, J.: On supporting interactive association rule mining. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, pp. 307–316. Springer, Heidelberg (2000)
16. Grudzinski, P., Wojciechowski, M.: Integration of candidate hash trees in concurrent processing of frequent itemset queries using Apriori. Control and Cybernetics 38, 47–65 (2009)
17. Grudzinski, P., Wojciechowski, M., Zakrzewicz, M.: Partition-Based Approach to Processing Batches of Frequent Itemset Queries. In: Larsen, H.L., Pasi, G., Ortiz-Arroyo, D., Andreasen, T., Christiansen, H. (eds.) FQAS 2006. LNCS (LNAI), vol. 4027, pp. 479–488. Springer, Heidelberg (2006)
18. Han, J., Fu, Y., Wang, W., Chiang, J., Gong, W., Koperski, K., Li, D., Lu, Y., Rajan, A., Stefanovic, N., Xia, B., Zaiane, O.: DBMiner: A System for Mining Knowledge in Large Relational Databases. In: Simoudis, E., Han, J., Fayyad, U.M. (eds.) Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, pp. 250–255. AAAI Press, Menlo Park (1996)
19. Han, J., Fu, Y., Wang, W., Koperski, K., Zaiane, O.: DMQL: A data mining query language for relational databases. In: Jagadish, H.V., Mumick, I.S. (eds.) Proceedings of the ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, pp. 27–33. ACM Press, New York (1996)
20. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Chen, W., Naughton, J.F., Bernstein, P.A. (eds.) Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 1–12. ACM Press, New York (2000)
21. Hart, J.P., Shogan, A.W.: Semi-greedy Heuristics: An Empirical Study. Operations Research Letters 6, 107–114 (1987)
22. Hipp, J., Guntzer, U.: Is pushing constraints deeply into the mining algorithms really what we want? - An alternative approach for association rule mining. ACM SIGKDD Explorations Newsletter 4, 50–55 (2002)
23. Imielinski, T., Mannila, H.: A Database Perspective on Knowledge Discovery. Communications of the ACM 39, 58–64 (1996)

24. Imielinski, T., Virmani, A.: MSQL: A Query Language for Database Mining. Data Mining and Knowledge Discovery 3, 373–408 (1999)
25. Imielinski, T., Virmani, A., Abdulghani, A.: Discovery board application programming interface and query language for database mining. In: Simoudis, E., Han, J., Fayyad, U. (eds.) Proceedings of the Second International Conference on Knowledge Discovery in Databases and Data Mining, pp. 20–26. AAAI Press, Menlo Park (1996)
26. ISO: Information technology – Database languages – SQL multimedia and application packages – Part 6: Data mining. ISO/IEC 13249-6 (2006)
27. Jain, S., Swamy, C., Balaji, K.: Greedy Algorithms for k-way Graph Partitioning. In: Sinha, P.K., Das, C.R. (eds.) Proceedings of the 6th International Conference on Advanced Computing., Tata McGraw Hill, New York (1998)
28. Jarke, M.: Common subexpression isolation in multiple query optimization. In: Kim, W., Reiner, D.S. (eds.) Query Processing in Database Systems, pp. 191–205. Springer, New York (1985)
29. Jedrzejczak, P., Wojciechowski, M.: Integrated Candidate Generation in Processing Batches of Frequent Itemset Queries Using Apriori. In: Fred, A., Filipe, J. (eds.) Proceedings of the 2nd International Conference on Knowledge Discovery and Information Retrieval, pp. 487–490. SciTePress (2010)
30. Jin, R., Sinha, K., Agrawal, G.: Simultaneous Optimization of Complex Mining Tasks with a Knowledgeable Cache. In: Grossman, R., Bayardo, R.J., Bennett, K.P. (eds.) Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 600–605. ACM Press, New York (2005)
31. JSR-73 Expert Group: Java Specification Request 73: Java Data Mining, JDM (2005)
32. Karypis, G.: Multilevel Hypergraph Partitioning. In: Cong, J., Shinnerl, J. (eds.) Multilevel Optimization Methods for VLSI. Kluwer Academic Publishers, Boston (2002)
33. Karypis, G., Kumar, V.: Multilevel Graph Partitioning Schemes. In: Banerjee, P., Boca, P. (eds.) Proceedings of the 24th International Conference on Parallel Processing, pp. 113–122. CRC Press, Boca Raton (1995)
34. Karypis, G., Han, E., Kumar, V.: Chameleon: A Hierarchical Clustering Algorithm Using Dynamic Modeling. IEEE Computer 32, 68–75 (1999)
35. Meo, R.: Optimization of a Language for Data Mining. In: Proceedings of the 2003 ACM Symposium on Applied Computing, pp. 437–444. ACM, New York (2003)
36. Meo, R.: Inductive Databases: Towards a New Generation of Databases for Knowledge Discovery. In: Proceedings of the First International Workshop on Integrating Data Mining, Database and Information Retrieval, pp. 1003–1007. IEEE Computer Society, Los Alamitos (2005)
37. Morzy, M., Wojciechowski, M., Zakrzewicz, M.: Optimizing a Sequence of Frequent Pattern Queries. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589, pp. 448–457. Springer, Heidelberg (2005)
38. Morzy, T., Wojciechowski, M., Zakrzewicz, M.: Data Mining Support in Database Management Systems. In: Kambayashi, Y., Mohania, M., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, pp. 382–392. Springer, Heidelberg (2000)
39. Morzy, T., Wojciechowski, M., Zakrzewicz, M.: Materialized Data Mining Views. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 65–74. Springer, Heidelberg (2000)
40. Nag, B., Deshpande, P.M., DeWitt, D.J.: Using a Knowledge Cache for Interactive Discovery of Association Rules. In: Han, J. (ed.) Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 244–253. ACM Press, New York (1999)

41. Netz, A., Chaudhuri, S., Fayyad, U., Bernhardt, J.: Integrating data mining with SQL databases: OLE DB for data mining. In: Proceedings of the 17th International Conference on Data Engineering, pp. 379–387. IEEE Computer Society, Los Alamitos (2001)
42. Ng, R., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained association rules. In: Tiwary, A., Haas, L.M. (eds.) Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, pp. 13–24. ACM Press, New York (1998)
43. Oracle Corporation: PL/SQL Packages and Types Reference, 10g Release 1 (10.1) (2003)
44. Pei, J., Han J.: Can We Push More Constraints into Frequent Pattern Mining? In: Ramakrishnan, R., Stolfo, S., Bayardo, R., Parsa, I. (eds.) Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 350–354. ACM Press, New York (2000)
45. Pei, J., Han, J., Lakshmanan, L.V.S.: Pushing Convertible Constraints in Frequent Itemset Mining. Data Mining and Knowledge Discovery 8, 227–252 (2004)
46. Roy, P., Seshadri, S., Sundarshan, S., Bhobe, S.: Efficient and Extensible Algorithms for Multi Query Optimization. In: Chen, W., Naughton, J.F., Bernstein, P.A. (eds.) Proceedings of 2000 ACM SIGMOD International Conference on Management of Data, pp. 249–260. ACM Press, New York (2000)
47. Sellis, T.K.: Multiple Query Optimization. ACM Transactions on Database Systems 13, 23–52 (1988)
48. Srikant, R., Vu, Q., Agrawal, R.: Mining association rules with item constraints. In: Heckerman, D., Mannila, H., Pregibon, D. (eds.) Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, pp. 67–73. AAAI Press, Menlo Park (1997)
49. Thomas, S., Bodagala, S., Alsabti, K., Ranka, S.: An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. In: Heckerman, D., Mannila, H., Pregibon, D. (eds.) Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, pp. 263–266. AAAI Press, Menlo Park (1997)
50. Wojciechowski, M., Galecki, K., Gawronek, K.: Three Strategies for Concurrent Processing of Frequent Itemset Queries Using FP-growth. In: Džeroski, S., Struyf, J. (eds.) KDID 2006. LNCS, vol. 4747, pp. 240–258. Springer, Heidelberg (2007)
51. Wojciechowski, M., Zakrzewicz, M.: Methods for Batch Processing of Data Mining Queries. In: Haav, H.-M., Kalja, A. (eds.) Proceedings of the 5th International Baltic Conference on Databases and Information Systems, Tallinn Technical University, pp. 225–236 (2002)
52. Wojciechowski, M., Zakrzewicz, M.: Dataset Filtering Techniques in Constraint-Based Frequent Pattern Mining. In: Hand, D.J., Adams, N.M., Bolton, R.J. (eds.) Pattern Detection and Discovery. LNCS (LNAI), vol. 2447, pp. 77–91. Springer, Heidelberg (2002)
53. Wojciechowski, M., Zakrzewicz, M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. In: Kalinichenko, L.A., Manthey, R., Thalheim, B., Wloka, U. (eds.) ADBIS 2003. LNCS, vol. 2798, pp. 76–87. Springer, Heidelberg (2003)
54. Wojciechowski, M., Zakrzewicz, M.: Data Mining Query Scheduling for Apriori Common Counting. In: Barzdins, J. (ed.) Proceedings of the 6th International Baltic Conference on Databases and Information Systems, University of Latvia, pp. 270–281 (2004)
55. Wojciechowski, M., Zakrzewicz, M.: Evaluation of the Mine Merge Method for Data Mining Query Processing. In: Benczur, A., Demetrovics, J., Gottlob, G. (eds.) Proceedings of the 8th East European Conference on Advances in Databases and Information Systems, Computer and Automation Research Institute, Hungarian Academy of Sciences, pp. 78–88 (2004)

56. Wojciechowski, M., Zakrzewicz, M.: On Multiple Query Optimization in Data Mining. In: Ho, T.-B., Cheung, D., Liu, H. (eds.) PAKDD 2005. LNCS (LNAI), vol. 3518, pp. 696–701. Springer, Heidelberg (2005)
57. Wojciechowski, M., Zakrzewicz, M.: Heuristic Scheduling of Concurrent Data Mining Queries. In: Li, X., Wang, S., Dong, Z.Y. (eds.) ADMA 2005. LNCS (LNAI), vol. 3584, pp. 315–322. Springer, Heidelberg (2005)
58. Wojciechowski, M., Zakrzewicz, M.: Partycjonowanie grafow a optymalizacja wykonania zbioru zapytan eksploracyjnych. In: Morzy, T., Rybinski, H. (eds.) Proceedings of I Krajowa Konferencja Naukowa Technologie Przetwarzania Danych, pp. 62–71. Wydawnictwo Politechniki Poznanskiej (2005)
59. Zakrzewicz, M., Morzy, M., Wojciechowski, M.: A Study on Answering a Data Mining Query Using a Materialized View. In: Aykanat, C., Dayar, T., Körpeoğlu, İ. (eds.) ISCIS 2004. LNCS, vol. 3280, pp. 493–502. Springer, Heidelberg (2004)