Data Access Paths in Processing of Sets of Frequent Itemset Queries

Piotr Jedrzejczak, Marek Wojciechowski

Poznan University of Technology, Piotrowo 2, 60-965 Poznan, Poland
Marek.Wojciechowski@cs.put.poznan.pl

Abstract. Frequent itemset mining can be regarded as advanced database querying where a user specifies the dataset to be mined and constraints to be satisfied by the discovered itemsets. One of the research directions influenced by the above observation is the processing of sets of frequent itemset queries operating on overlapping datasets. Several methods of solving this problem have been proposed, all of them assuming selective access to the partitions of data determined by the overlapping of queries, and tested so far only on flat files. In this paper we theoretically and experimentally analyze the influence of data access paths available in database systems on the methods of frequent itemset query set processing, which is crucial from the point of view of their possible applications.

1 Introduction

Frequent itemset mining [1] is one of the fundamental data mining techniques, used both on its own and as the first step of association rules generation. The problem of frequent itemset and association rule mining was initially formulated in the context of market-basket analysis, aiming at the discovery of items frequently co-occurring in customer transactions, but it quickly found numerous applications in various domains, such as medicine, telecommunications and World Wide Web.

Frequent itemset mining can be regarded as advanced database querying where a user specifies the source dataset, the mininum support threshold and (optionally) the pattern constraints within a given constraint model [7]. Frequent itemset queries are therefore a special case of data mining queries.

Many frequent itemset mining alogrithms have been developed. The two most prominent classes of algorithms are determined by the strategy of the pattern search space traversal. Level-wise algorithms, represented by the classic *Apriori* algorithm [3], follow the breadth-first strategy, whereas pattern-growth methods, among which *FP-growth* [6] is the best known, perform the depth-first search.

Although many algoritms have been proposed, effective knowledge discovery in large volumes of data remains a complicated task and requires considerable time investment. Long data mining query execution times often result in queries being collected and processed in a batch when the system load is lower. Since those queries may have certain similarities, e.g. refer to the same data, processing

them concurrently rather than sequentially gives the opportunity to execute the whole set of queries much more effectively [10].

As far as processing batches of frequent itemset queries is concerned, several methods exploiting the overlapping of queries' source datasets have been developed: *Mine Merge*, independent of the frequent itemset mining algorithm used [10]; *Common Counting* [10] and *Common Candidate Tree* [5] designed for *Apriori*; *Common Building* and *Common FP-tree* [11] based on *FP-growth*. Each of these methods, in addition to the theoretical analysis, has been tested in practice with the use of flat files and direct access paths to the source dataset's partitions. In reality, however, the mined data is often stored in databases, where, depending on the selection conditions, many different access paths may be available.

The aim of this paper is both the theoretical and practical analysis of the aforementioned concurrent frequent mining methods in the light of different data access paths. As the *FP-growth* methods are adaptions of the methods developed for *Apriori*, the analysis will be conducted for *Apriori* only. *Apriori* it is the most widely implemented frequent itemset mining algorithm and the multiple source data reads it performs should make the differences between the access paths and their impact on the total execution time more noticeable.

The topics discussed in this paper can be regarded as multiple-query optimization, which was previously extensively studied in the context of database systems [9] geared towards building a global execution plan that exploits the similarities between queries. In the field of data mining, except the problem discussed in this paper, multiple-query optimization was considered in a vastly different problem of frequent itemset mining in multiple datasets [8]. Solutions similar to the ones in this paper, however, can be found in the related domain of logic programming, where a method similar to *Common Counting* has been proposed [4].

2 Multiple-Query Optimization for Frequent Itemset Queries

2.1 Basic Definitions and Problem Statement

Itemset. Let $I = \{i_1, i_2, ..., i_n\}$ be a set of literals called *items*. An *itemset* X is a set of items from I, ie. $X \subseteq I$. The *size* of the itemset X is the number of items in it.

Transaction. Let D be a *database* of transactions, where *transaction* T is a set of elements such that $T \subseteq I$ and $T \neq \emptyset$. A transaction T *supports* the item $x \in I$ if $x \in T$. A transaction T *supports* the itemset $X \subseteq I$ if it supports all items $x \in X$, ie. $X \subseteq T$.

Support. The *support* of the itemset X in the database D is the number of transactions $T \in D$ that support X.

Frequent itemset. An itemset $X \subseteq I$ is *frequent* in D if its support is no less than a given *minimum support* threshold.

Frequent itemset query. A *frequent itemset query* is a tuple $dmq = (R, a, \Sigma, \Phi, minsup)$, where R is a database relation, a is a set-valued attribute of R, Σ is a condition involving the attributes of R called *data selection predicate*, Φ is a condition involving discovered itemsets called *pattern constraint*, and *minsup* 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 $\geq$ *minsup* (π and σ denote relational projection and selection operations respectively).

Elementary data selection predicates. 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 $\sigma_{s_u} R \cap \sigma_{s_v} R = \emptyset$ and for each dmq_i there exist integers $a, b, ..., m$ such that $\sigma_{\Sigma_i} R = \sigma_{s_a} R \cup \sigma_{s_b} R \cup .. \cup \sigma_{s_m} R$. The set of elementary data selection predicates represents the partitioning of the database determined by overlapping of queries' datasets.

Problem. 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.

2.2 Apriori

Introduced in [3] and based on the observation that every subset of a frequent itemset is also frequent, the *Apriori* algorithm iteratively discovers frequent itemsets of increasing size. Frequent 1-itemsets are discovered by simply counting the occurences of each item in the database. Following iterations consist of two phases: the generation phase, during which frequent itemsets from previous iteration are used to generate candidate itemsets of size 1 more, and a verification phase, during which the algorithm counts the occurences of those itemsets in the database and discards the ones that do not meet the minimum support threshold. This process is repeated until no more frequent itemsets are discovered. To avoid performing a costly inclusion test for every candidate and every read transaction, generated candidate itemsets are stored in a hash tree.

3 Review of Existing Methods

3.1 Sequential Execution

The simplest way of processing a set of frequent itemset queries is to process them sequentially using a standard algorithm like the aforementioned *Apriori*. This represents the naive approach and even though it's not an effective solution to the problem, it provides a natural reference point when evaluating other methods.

3.2 Common Counting

The *Common Counting* [10] method reduces the amount of required data reads by integrating the scans of those parts of the database that are shared by more

than one query. All queries from the set are executed concurrently in a two-phase iterative process similar to *Apriori*. Candidate generation is performed seperately for each query, with the generated candidates stored in separate hash trees. Verification, however, is performed simultaneously for all queries during a single database scan. Each database partition is therefore read only once per iteration, effectively reducing the number of I/O operations.

3.3 Common Candidate Tree

While *Common Counting* optimizes only the database reads, *Common Candidate Tree* [5] goes a step further and shares the data structures between the concurrently processed queries as well. Like in *Common Counting*, each partition is read only once per iteration, but this time a single hash tree is shared by all queries from the set, reducing the cost associated with inclusion tests. While the structure of the hash tree itself remains identical to the one used in the original *Apriori*, the candidate itemsets are modified to include a vector of counters (one for each query) and a vector of boolean flags (to track which queries generated the itemset). Candidate generation is performed seperately for each query as in *Common Counting*, with the generated sets of candidates being merged into the extended representation and put in the common hash tree afterwards. Only that single tree is then used during the verification phase, with only the appropriate counters (ie. those corresponding to queries that both generated the candidate and refer to the currently processed partition) being incremented.

3.4 Mine Merge

The *Mine Merge* [10] algorithm presents an entirely different approach. It employs the property that in a database divided into partitions, an itemset frequent in the whole database is also frequent in at least one of the partitions. *Mine Merge* first generates intermediate queries, each of them based on a single elementary data selection predicate (ie. each referring to a single database partition). Intermediate queries are then executed sequentially (for example using *Apriori*) and their results are used to create a global list of candidate itemsets[1] for each original query. A single database scan is then performed to calculate the support of every candidate itemset and discard the ones below the desired threshold, thus producing the actual results for each of the original queries.

4 Data Access Paths in Frequent Itemset Query Set Processing

4.1 Data Structures and Access Paths

In today's world the vast majority of data, including the data targeted by frequent itemset mining, is stored in relational database systems. Contemporary

[1] Such a list consists of frequent itemsets from all partitions the query refers to.

relational database management systems (DBMSs) follow the SQL standard and offer similar fundamental functionality in the sense that they store data in tables, which can be accompanied by indexes to speed-up the selection of that data. Thus, as far as the analyzed frequent itemset query processing methods are concerned we can generally assume that: (1) the data to be mined is stored in a database table, (2) each data selection predicate selects a subset of rows from that table, (3) there are two methods of accessing the rows satisfying a given data selection predicate: a full scan of the table during which the predicate is evaluated for each row, and selective access with the help of index structures.

While keeping the analysis as general and product-independent as possible, it should be noted that DBMSs available on the market compete with each other and therefore provide different choices for table organization and indexing. In the experiments accompanying our theoretical study we use Oracle 11g, considered the industry-leading database management system. Oracle 11g is an example of an object-relational DBMS, i.e. it offers object extensions to the relational model such as user-defined types and collections. We use a VARRAY collection type to store itemsets. The default table organization in Oracle 11g is heap (which is unordered). Two types of indexes are available: B-tree and bitmap indexes. We use B-trees as they support range selection predicates. An interesting alternative to a heap-organized table accompanied by an index in Oracle 11g is an index-organized table. We also consider it in the experiments.

4.2 Implementation of Compared Methods

All the compared methods were formulated in terms of reading partitions corresponding to elementary data selection predicates. Therefore, if all partitions of the table can be selectively accessed thanks to an index, all the considered methods are directly applicable with no need for extra optimizations. The question is whether these methods can avoid performing a separate full scan of the table for each partition if no applicable index is available or the query optimizer decides not to use it[2].

As for *Mine Merge*, we currently do not see any satisfactory solutions that would prevent it from suffering a significant performance loss when full scans are necessary[3]. However, it should be noted that since *Mine Merge* executes its intermediate queries independently of each other, each query can employ a different access path (a full scan or an index scan depending on the index availability and the estimated cost).

Contrary to *Mine Merge*, *Common Counting* and *Common Candidate Tree* can be implemented in a way that minimizes the negative effects of full scans.

[2] The optimizer might not use an index if a full scan results in a lower estimated cost due to poor selectivity of the index for a given selection predicate. In our discussion it is not relevant what the reason for performing a full scan to access a partition actually was.

[3] One possible solution is to materialize partitions corresponding to intermediate queries in one full table scan, but we consider it impractical for large datasets.

Both methods read multiple partitions (the ones referred to by the queries still being executed) per iteration. However, since their candidate counting is in fact performed per transaction, not per partition (individual transactions passed through hash trees), the actual order in which transactions are retrieved from the database is irrelevant. Thus, *Common Counting* and *Common Candidate Tree* can perform a single SQL query in each iteration, reading the sum of the partitions required by the queries whose execution still did not finish. This modification is crucial if full scans would be required to retrieve any individual partition (one full scan instead of several full scans and/or table accesses by index per iteration[4]) but can also be beneficial if all the partitions are accessible by index (in certain circumstances reading all the partitions in one full table scan may be more efficient than reading them one by one using an index[5]).

4.3 Theoretical Cost Analysis

In order to analyze the impact of data access paths we will provide cost formulas for the amount of data read by the compared methods for both selective access and full table scans. We will not include the cost of in-memory computations in the formulas as it does not depend on the chosen data access path. For the sake of simplicity we will assume that all *Apriori* executions (for the original as well as *Mine Merge* intermediate queries) require the same number of iterations. The variables appearing in the formulas are as follows: k - the number of *Apriori* iterations for each query, n - the number of original queries, ni - the number of *Mine Merge* intermediate queries, DB - the size of the database table containing input data, SUM - the sum of the sizes of the original queries' datasets, CVR - the total size of the parts of the table referred to (covered) by the queries.

The cost formulas for sequential execution for selective access (SEQ_{IDX}) and full table scans (SEQ_{FULL}) are presented below. For selective data access each query reads its source dataset k times. With full scans each query reads the whole table k times.

$$SEQ_{IDX} = k * SUM, \qquad SEQ_{FULL} = n * k * DB \tag{1}$$

The formulas for *Mine Merge* include the cost of the (additional) verifying scan of data. Full scan formula involves the number of intermediate queries, which is not present in the formula for selective data reads – in that case, only the amount of covered data is important, not the number of partitions into which it is divided.

$$MM_{IDX} = (k + 1) * CVR, \qquad MM_{FULL} = (ni * k + 1) * DB \tag{2}$$

Common Counting and *Common Candidate Tree* differ only in in-memory data structures, therefore the two methods share the formulas for data access

[4] Even if for just one partition a full scan is the only or the best option, all the partitions are to be retrieved in one full table scan. This is different from Mine Merge where each partition could be retrieved using a different access path.

[5] The choice of an access path is up to the query optimizer.

costs. Thanks to the integrated full scan proposed in Sect. 4.2, the cost for full scans does not depend on the number of queries (similarly as in the case of selective access).

$$CC_{IDX} = k * CVR, \qquad CC_{FULL} = k * DB \tag{3}$$

When comparing the above cost formulas one should take into account that: $n * DB \geq SUM \geq CVR$, $DB \geq CVR$, $ni \geq n$ (regarding the latter, the upper limit on ni is $2^n - 1$)[6].

Comparing the data access costs per algorithm, the increase of the cost when selective access is replaced with full scans varies among the methods: it is the smallest for *Common Counting* (independent of the number of queries) and the biggest for *Mine Merge* (dependent on the number of intermediate queries). The consequence of the above difference is a questionable applicability of *Mine Merge* if the data has to be retrieved using full table scans. With selective access *Mine Merge* should outperform sequential execution, provided the overlapping among the queries (exploited in each *Apriori* iteration) compensates for the extra scan of data[7]. With full scans *Mine Merge* can be expected to always perform worse than sequential execution. On the other hand, *Common Counting* (and *Common Candidate Tree*) not only should outperform sequential execution regardless of the available access path but even relatively benefit from full scans.

5 Experimental Results

Experiments were conducted on a synthetic dataset generated with GEN [2] using the following settings: number of transactions = 10 000 000, average number of items in a transaction = 8, number of different items = 1 000, number of patters = 15 000, average pattern length = 4. Data was stored as `<transaction id, varray of item>` pairs inside Oracle 11g database deployed on SuSE Linux, with the test application written in Java running on Mac OS X 10.6.6. Database connection was handled through JDBC over 1 Gigabit Ethernet.

Two experiments were conducted: the first one included two fixed-size queries with varying level of overlapping between them; in the second the overall scope of the processed part of the dataset was fixed while the number of fixed-size queries in the set varied. Both experiments measured the execution times of sequential execution (SEQ), *Common Counting* (CC), *Common Candidate Tree* (CCT) and *Mine Merge* (MM) for both the sequential (full scan) and selective (index scan[8]) access paths.

[6] The upper limit on the number of *Mine Merge* intermediate queries (equal to the number of elementary data selection predicates) can be smaller if certain constraints on data selection predicates are applied. For example, if all the predicates select single ranges of the same attribute, the maximal number of intermediate queries is $2 * n - 1$.

[7] In our analysis we do not consider the differences in the cost of in-memory computation, which can be a differentiator if the data access costs are identical or similar.

[8] The same experiments were repeated using an index-organized table, giving consistent results.

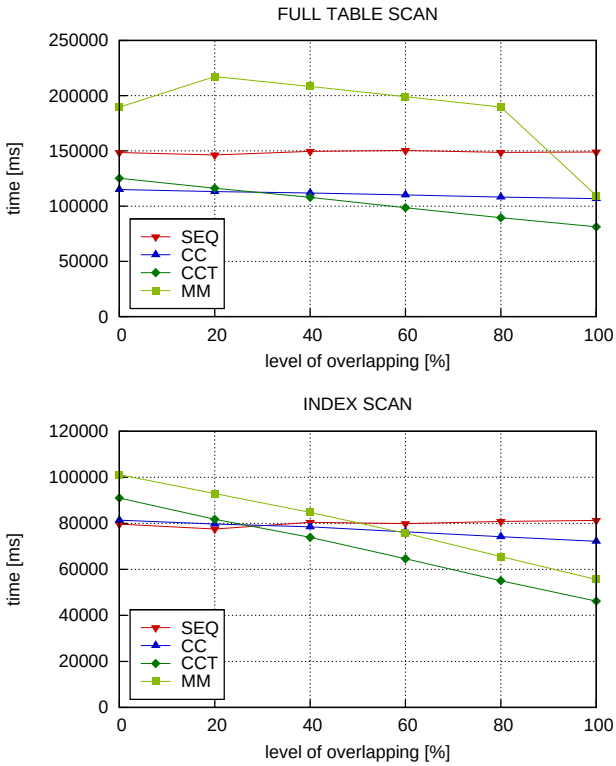

Fig. 1. Execution times for two queries and different levels of overlapping.

The results of the first experiment for two queries of 1 000 000 transactions each, minimum support of 0.7%[9] and the level of overlapping from 0% to 100% are shown in Fig. 1.

As predicted, *Mine Merge* performed significantly worse than other methods without selective access, losing even with the sequential execution. With index scans available its loss wasn't as noticeable and it even managed to outperform *Common Counting* when the level of overlapping is high enough.

Both *Common Counting* and *Common Candidate Tree* performed well regardless of the access path. While their times for lower levels of overlapping were similar, *Common Candidate Tree* was clearly better when queries overlapped signficantly.

The second experiment had the queries access the same fixed part of the database each time. The query set consisted of 2 to 6 queries of size 600 000

[9] Experiments were conducted with two different minimum support thresholds of 0.7% and 2% with consistent results; due to limited space, only the former threshold is presented.

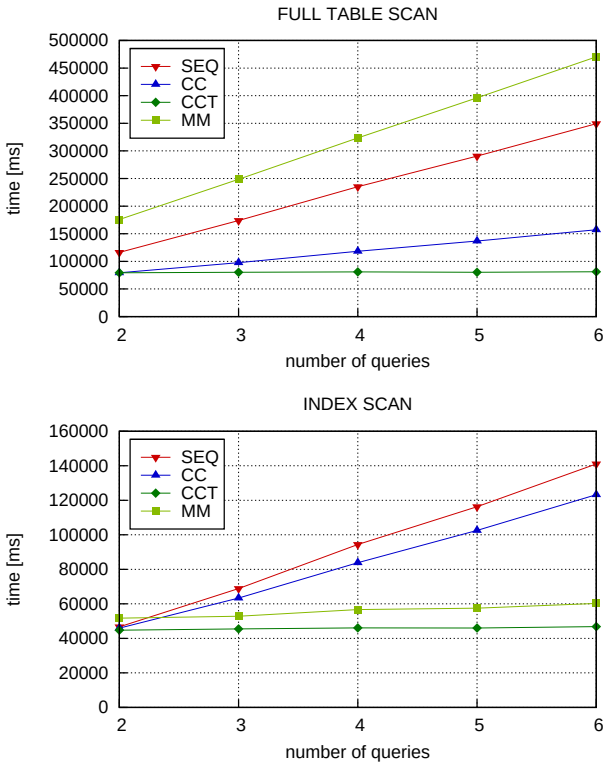

Fig. 2. Execution times for fixed scope and different numbers of queries.

transactions each, spread evenly across the first 1 000 000 transactions from the database (each time the first query in the set referred to transactions with identifiers from 0 to 600 000, the last one – 400 000 to 1 000 000). Results are presented in Fig. 2.

As was the case in the first experiment, *Mine Merge* was very inefficient when forced to execute full table scans, performing even worse than sequential execution. With selective access, however, the number of queries had little impact on *Mine Merge* execution times, which again allowed it to perform better than *Common Counting* and quite close to *Common Candidate Tree*, which was the fastest algorithm for both access paths. *Common Counting*, though better than sequential execution in both cases, provided a more noticeable gain over the naive method during full scans than when using the selective access path.

6 Conclusion

We considered the influence of data access paths available in DBMSs on the implementations and performance of the methods of frequent itemset query set

processing designed for the *Apriori* algorithm. As expected, both the theoretical and experimental analysis showed that the performance of all the compared methods suffers if selective access to data partitions is replaced with full scans. However, an important conclusion is that while the negative effect of full scans on *Mine Merge* is more significant than in the case of sequential processing, properly implemented *Common Counting* and *Common Candidate Tree* actually increase their advantage over sequential execution if full scans are necessary. In other words, *Mine Merge* is strongly dependent on efficient access paths to data partitions, whereas *Common Counting* and *Common Candidate Tree* can be successfully applied regardless of available data access paths.

References

1. Agrawal, R., Imielinski, T., Swami, A.N.: 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 (1993)
2. Agrawal, R., Mehta, M., Shafer, J.C., Srikant, R., Arning, A., Bollinger, T.: The quest data mining system. In: Simoudis, E., Han, J., Fayyad, U.M. (eds.) Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining. pp. 244–249. AAAI Press (1996)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) Proceedings of the 20th Int. Conf. on Very Large Data Bases. pp. 487–499. Morgan Kaufmann (1994)
4. 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)
5. Grudzinski, P., Wojciechowski, M.: Integration of candidate hash trees in concurrent processing of frequent itemset queries using apriori. In: Proceedings of the 3rd ADBIS Workshop on Data Mining and Knowledge Discovery. pp. 71–81 (2007)
6. 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 (2000)
7. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Communications of the ACM 39(11), 58–64 (1996)
8. 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 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 600–605. ACM (2005)
9. Sellis, T.K.: Multiple-query optimization. ACM Transactions on Database Systems 13(1), 23–52 (1988)
10. Wojciechowski, M., Zakrzewicz, M.: Methods for batch processing of data mining queries. In: Proceedings of the 5th International Baltic Conference on Databases and Information Systems. pp. 225–236 (2002)
11. Wojciechowski, M., Galecki, K., Gawronek, K.: Three strategies for concurrent processing of frequent itemset queries using fp-growth. In: Dzeroski, S., Struyf, J. (eds.) Knowledge Discovery in Inductive Databases. 5th Int'l Workshop, KDID 2006 Berlin, Germany, September 18, 2006 Revised Selected and Invited Papers. pp. 240–258. Springer (2007)