International Journal of Database Theory and Application
Vol.6, No.6 (2013), pp.63-74
http://dx.doi.org/10.14257/ijdta.2013.6.6.06

Mining Periodic Workload Patterns in Database Audit Trails

Marcin Zimniak[1]
marcin.zimniak@cs.tu-chemnitz.de

Janusz R. Getta[2]
jrg@uow.edu.au

Wolfgang Benn[1]
benn@cs.tu-chemnitz.de

[1] Faculty of Computer Science, TU Chemnitz, Germany
[2] School of Computer Science and Software Engineering,
University of Wollongong, Australia

Abstract

Information about periodic processing of database operations has a pivotal importance for continuous physical database design and automated performance tuning of database systems. This work shows how to detect the oscillations of database workloads caused by the periodical invocations of user applications. In particular, we present an algorithm for discovering periodic patterns in the histories of processing of complex and elementary database operations. In our approach, information collected from the database audit trails is transformed into a sequence of syntax trees and later on it is compressed in a syntax tree table. The periodic patterns are discovered through nested iterations over a four dimensional space of syntax trees and positional parameters of the patterns. Transformations of the patterns are used to discover the overlaping periodic patterns.

Keywords: periodic pattern, database audit trail, automated performance tuning

1 Introduction

It is well known that database workloads oscillate in time. The oscillations are caused by the periodical invocations of database applications, which process data on behalf of human operators, like for example customers accessing bank accounts, students enrolling courses, stock exchange broker performing financial operations on a stock market, etc. The periodic iterations of real world processes reflect on a database system as the periodic changes in its workload. The variations of workload levels can be discovered from historical information stored in the log files, traces from processing of database applications, audit trails, etc. Automated performance tuning of database systems [5] requires the prognostics on the variations of future database workload as well as the frequencies with which the user applications access the data containers. In a typical scenario, information about a period of low database workload and about data containers to be accessed in the future allows for appropriate restructuring of data containers to speed up their future processing. For example, a period of time when a database workload is low can be used to index data containers, pin data containers in data buffer caches, partition data containers, etc.

ISSN: 2005-4270 IJDTA
Copyright ⓒ 2013 SERSC

At the first glance the problem of discovering periodic patterns in the database workloads seems to be very similar to the classical problem of periodicity mining in the long sequences of numerical data such as time series [12],[6] or in the long sequences of genetic information [8]. The algorithms that find periodicity in time series categorize the elements of time series into a number of ranges and associate a timestamp with each value. A history of database workload is a sequence of data processing statements, like for example SQL statements in relational database system, collected by a database administrator over the long periods of time. This makes input data structures similar to time series as each data processing statement is also associated with a timestamp. However, due to the internal structures of complex data processing statements the problem cannot be treated in the same way as analysis of atomic data elements in time series or genetic sequences. A data processing statement is an expression over the elementary database operations and the data containers as the arguments of the operations. Analysis of database workload must consider the frequencies of elementary database operations as well as the complex ones that represent complete data processing statements. For example, if a number of SQL statements shares a common subquery then it may happen that sub query will reveal a periodic pattern even though the individual SQL statements will not. The next important difference is a choice of time units over which the periodical processing of the same data operations is performed. The traditional approaches assume fixed size and adjacent time units and fixed length of discovered patterns. In our case, the cycles are pretty well determined by the real world events that happen in daily, monthly and yearly workload of a database system. For example, backups are taken regularly, the contents of batch queue jobs is processed regularly, archiving is processed regularly, etc. Periodically processed user applications have the frequencies consistent with the frequencies of real world events. Such observations allows for discovery of periodic data processing which do not have the same frequencies over a given period of time. For example, the students enroll courses at the beginning of academic sessions, customer purchase tickets periodically occurring sport event, etc.

The problem of discovering periodic patterns in the database workloads is also quite similar, however, it is not not exactly the same as a problem of mining cyclic association rules [11]. Mining of cyclic association rules looks at the periodic executions of the largest sets of items that have enough support. For example, a cyclic association rule may say that applications a_i and a_j are computed in more or less the same period of time every day just after midnight. Mining of cyclic association rules tries to find the largest sets of operations on data that are periodically processed by a database system. In our case the largest sets of operations do not necessarily mean the highest workload. Sometimes a single periodically processed application issues a complex query that significantly contributes to a database workload. Additionally, mining of cyclic association rules is not able to discover two or more periodic periodic patterns whose cycles overlap on the same period of time. Such case happens when a user application is periodically processed with different frequencies by two or more users. In transaction processing the total number of operations is not as important from performance point of view as the total number, size and frequency of concurrently accessed data items. A set of data operations does not provide as precise information about the predicted database workload as sequences of operations which determine an order in which data containers are accessed.

In this work, we consider an environment of relational database system where the user applications submit SQL statements for processing in a database. A specific objective is to detect the periodical repetitions of expressions of extended relational algebra over the

Copyright ⓒ 2013 SERSC

fixed size time units. To discover the periodic patterns in the processing of elementary and complex database operations we use information about the past "behaviour" of a database system stored in the anonymized *audit trails*. An audit trail contains information about the scopes of individual database applications and about SQL statements processed by the applications. Database auditing allows to target the given groups of users performing operations on the given groups of data containers. We analyze an audit trail in order to detect the applications which require significant processing time and the high amounts of consumed resources. SQL statements obtained from an audit trail are later on processed with `EXPLAIN PLAN` statement in order to obtain the precise execution plans and estimation of processing costs. An execution plan is an expression built over the names of data containers and data processing operations such as sequential read and filtering of a relational table, vertical traversal of an index, hash implementation of join operation and the others. Next, the execution plans are converted into the syntax trees and saved into a syntax tree table. The table is further reduced and later on it is used by the iterations that reveal the periodic patterns in the processing of database operations.

The paper is organized in the following way. The next section reviews the major works on periodicity mining in time series and mining cyclic association rules. Section 3 defines an environment of relational database and the concepts of audit trail, syntax tree table, and time units. A concept of periodic patterns in database workload is introduced in Section 4. An algorithm for discovering periodic patterns in audit trails is explained in Section 5. Finally, Section 6 concludes the paper.

2 Related work

Easily available historical information on the performance aspects of database systems allows for application of data mining techniques to discover the periodic patterns in the database workloads [9], [13]. Sequences of operations on data recorded along the various periods of time can be easily described by the temporal predicates within a formal scope of Temporal Programming Logic and temporal deductive database systems [3], [1] Data mining techniques that inspired the works on period patterns came from the works mining frequent itemsets/association rules [2] and later on mining frequent episodes [10] and its extensions on mining complex episodes [15].

A work [11] was a starting point to many works on discovering cyclic patterns. It defined the principle concepts of cycle pruning, cycle skipping, cycle elimination heuristics.

The problem of discovering cyclic patterns seems to be very similar to a typical periodicity mining in time series [12],[6] where analysis is performed on the long sequences of elementary data items discretized into a number of ranges and associated with the timestamps. In our case, input data is a sequence of complex data processing statements, like for example SQL statements, which due to its internal structure cannot be treated in the same way as analysis of elementary data elements in time series or genetic sequences. The complex data processing statement forms a lattice[4] whose elements are syntax trees of the statements with a partial order determined by an inclusion relationship on syntax trees [14].

In the recent years more work on discovering period patterns addressed full periodicy, partial periodicity, perfect and imperfect periodicity [7] and recently asynchronous periodicity [16],[17],[18].

Our problem is also similar to a problem of mining cyclic association rules [11] where

Copyright © 2013 SERSC

an objective is to find the periodic executions of the largest sets of items that have enough support. However, in our case the largest sets of operations do not necessarily mean the highest workload and sometimes a single periodically processed application significantly contributes to a database workload.

Invocation of operation on data along the various points in time can be easily described by temporal predicates within a formal scope of Temporal Programming Logic and temporal deductive database systems [3], [1]. The reviews of data mining techniques based on analysis of ordered set of operations on data performed by the user applications ara available in [9], [13]. The model of periodicity considered in this paper is consistent with the model proposed in [19].

3 Database processing model

In this work, we consider a typical relational database system where a relational model of data is used to represent data containers. Let x be a nonempty set of attribute names later on called as a *schema* and let $dom(a)$ denotes a domain of attribute $a \in x$. A *tuple* t defined over a schema x is a full mapping $t : x \rightarrow \cup_{a \in x} dom(a)$ and such that $\forall a \in x$, $t(a) \in dom(a)$. A *relational table* r created on a schema x is a set of tuples over a schema x. A query processor transforms SQL statements submitted by the user applications into the query execution plans formulated as the expressions of extended relational algebra. Processing of SQL statements is recorded in a database *audit trail*.

3.1 Audit trail

A history of SQL processing is stored in a trace from processing of *user applications* $a_1, \ldots, a_n$. A *trace* of a user application a_i is a finite sequence of pairs $<c_i{:}t_{c_i},\ s_{i_1}{:}t_{i_1},\ \ldots,\ s_{i_n}{:}t_{i_n},\ d_i{:}t_{d_i}>$ where c_i is a *connect* statement, t_{c_i} is a timestamp when the statement has been processed, all s_{i_j} are SQL statements, all t_{i_j} are timestamps of the respective SQL statements, d_i is a *disconnect* statement, and t_{d_i} is a timestamp of disconnect statement. Processing of an application a_i starts from processing of a connect statement c_i, then it follows with processing of SQL statements s_{i_j}, and it finally ends with processing of a disconnect statement d_i. An *audit trail* is complete trace from processing of many concurrently running user applications. Due to concurrent processing of the applications, an audit trail is an interleaved sequence of connect, disconnect, and SQL statements. For example, a sequence $<c_i{:}t_{c_i}, s_{i_1}{:}t_{i_1}, c_j{:}t_{c_j}, s_{j_1}{:}t_{j_1}, s_{i_2}{:}t_{i_2}, d_i{:}t_{d_i}\ d_j{:}t_{d_j}>$ is a sample audit trail from the processing of applications a_i, and a_j.

Information about processing of a statement s_{i_j} is associated in an audit trail with a timestamp t_{i_j}. SQL statements can be easily extracted from an audit trail and `EXPLAIN PLAN` statement can be used in the contexts of respective user schemas to transform the statements into the syntax trees of query execution plans over a set of operations of extended relational algebra. The codes of operations of extended relational algebra are used as the labels of nonleaf nodes in syntax trees and the names of data containers processed by the operations are used as the labels of nonleaf nodes. The operations of extended relational algebra include the implementation dependent variants of operations of standard relational algebra such as *selection*, *projection*, *join*, *antijoin*, set operation, and operations of *grouping*, *sorting*, and aggregate functions. Due to the different implementation techniques, the

 Copyright ⓒ 2013 SERSC

operations from the basic system of relational algebra, e.g. *selection* or *join* contribute to an number of different elementary operations depending on their implementations, e.g. *index based selection*, *full scan selection*, *hash based join*, *index based join*, etc.

3.2 Syntax tree table

Let s_i and s_j be the statements obtained from an audit trail and let T_{s_i}, T_{s_j} be their respective syntax trees obtained from the applications of `EXPLAIN PLAN` statement. We say that syntax trees T_{s_i}, T_{s_j} are the same if the labels in root nodes of the trees are the same the respective subtrees directly connected to the root nodes are the same or the labels of leaf level nodes connected to root node are the same.

We say, that a syntax tree T_{s_i} is included in or equal to a syntax tree T_{s_j} and we denote it with $T_{s_i} \sqsubseteq T_{s_j}$ if there exists a nonleaf node n in a syntax tree T_{s_j} such that a subtree with a root node n is the same as a syntax tree T_{s_i}.

Complete information about syntax trees of SQL statements extracted from an audit trail is stored in an *syntax tree table*. A syntax tree table is a set of tuples <*tree*, *operation*, $left$, *right*, *workload*, *timestamps*> where *tree* is a unique identifier of a syntax tree, *operation* is a code of extended relational algebra operation at the root of syntax tree identified by *tree*, $left$ and *right* are the identifiers of left and right argument of syntax tree identified by *tree* or the names of relational tables, *workload* is an estimate workload imposed on a database system when processing a syntax tree *tree*, and *timestamps* is a set of all timestamps when a syntax tree *tree* was processed by a database system. A syntax tree table is created such that each syntax tree is stored in the table only once no matter how many times it was included in the processed syntax trees. We shall say that a subtree t_{leaf} is a *leaf level subtree* of a syntax tree T_s if $t_{leaf} \sqsubseteq T_s$ and both arguments of an operation in a root node of t_{leaf} are data containers. A syntax tree table is created in the following way.

(1) Make a syntax tree table empty

(2) For all syntax trees obtained from audit trail repeat the following actions.

(2.1) Let T_s be syntax tree of a statement s. For all leaf level subtrees in T_s repeat the following actions.

(2.1.1) Let t_{leaf} be a leaf level subtree in T_s. Search a syntax tree table for a tuple that has a value of *code* equal to an operation code in a root node of t_{leaf} and $left$ equal to the left argument of t_{leaf} and *right* equal to the right argument of t_{leaf}

(2.1.2) If a tuple searched for is found then that a subtree the same as t_{leaf} has been already re-ordered in the table. Then add a timestamp of t_{leaf} to *timestamps* in the tuple found.

(2.1.3) If a tuple searched for is not found then create and append a new tuple to a syntax tree table. The new tuple should obtain a new automatically generated *tid*. A code of operation in a root of t_{leaf} becomes a value of *code*. The parameters $left$ and *right* obtain the values of left and right arguments of t_{leaf}. A value of workload imposed by the computation of t_{leaf} becomes a value of *workload* and finally *timestamps* becomes a single element set that consist of a timestamp of t_{leaf}.

(2.1.4) A subtree t_{leaf} is removed from T_S such that a root node of t_{leaf} is replaced either with a *tid* found in step (2.1.2) if t_{leaf} has been already recorded in syntax tree table

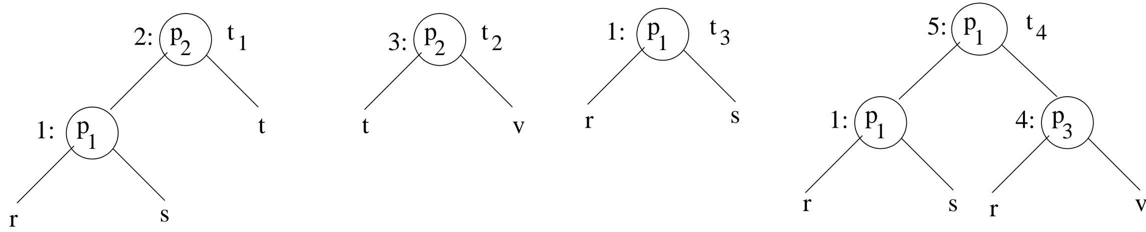

Figure 1. A sequence of syntax trees

Table 1. A sample syntax tree table

tree	*operation*	*left*	*right*	*workload*	*timestamps*
1	p_1	r	s	w_1	$\{t_1, t_3, t_4\}$
2	p_2	1	t	w_2	$\{t_1\}$
3	p_2	t	v	w_3	$\{t_2, t_4\}$
4	p_3	r	v	w_4	$\{t_4\}$
5	p_1	1	4	w_5	$\{t_4\}$

or a the new *tid* found in a step (2.1.3) if a new tuple representing t_{leaf} has been inserted into a syntax tree table.

(2.1.5) If T_s still has at least one leaf level subtree then return to step (2.1.1).

(2.2) If there is at least one more syntax tree top be stored in a syntax tree table then return to step (2.1). Otherwise all syntax trees from an audit trail has been stored in a syntax tree table.

As a simple example consider a sequence of syntax trees processed at the timestamps t_1, t_2, t_3, and t_4 given in Fig. 1 where p_1, p_2, and p_3 are the codes of operations. The respective syntax tree table is given below.

3.3 Time units

Let $<t_{start}, t_{end}>$ be a period of time over which an audit trail is recorded. The period is divided into a contiguous sequence of disjoint and fixed size *elementary time units* $<t_e^{(i)}, \tau_e>$ where $t_e^{(i)}$ for $i = 1, \ldots, n$ is a timestamp when an elementary time unit starts and τ_e is a length of the unit. Elementary time units are distributed over $<t_{start}, t_{end}>$ such that $t_{start} = t_e^{(1)}$ and $t_e^{(i+1)} = t_e^{(i)} + \tau_e$ and $t_e^{(n)} + \tau_e = t_{end}$.

A *time unit* is a pair $<t, \tau>$ where t is a start point of a unit and τ is a length of the unit. A time unit consists of one or more consecutive elementary time units.

A sequence U of n disjoint time units $<t^{(i)}, \tau^{(i)}>$ $i = 1, \ldots, n$ over $<t_{start}, t_{end}>$ is any sequence of time units that satisfies the following properties: $t_{start} \leq t^{(1)}$ and $t^{(i)} + \tau^{(i)} \leq t^{(i+1)}$ and $t^{(n)} + \tau_{(n)} \leq t_{end}$.

As a simple example consider an audit trail that starts on $t_{01-01-20070.00am}$ and ends on $t_{31-01-200712.00pm}$. Then, a sequence of disjoint time units called as *morning tea time* consists of the following units $<t_{01:01:2007:10:30am}, 30mins>$, $<t_{02:01:2007:10:30am}, 30mins>$, $\ldots$, $<t_{31:01:2007:10:30am}, 30mins>$.

 Copyright © 2013 SERSC

4 Periodic patterns

In this section we define a concept of *periodic pattern* and its validation in an audit trail. A *periodic pattern* is a tuple $<T_s, U, l, b, e, p>$ where T is a syntax tree of a statement s, U is a sequence of disjoint time units over which the pattern occurs, l is a threshold load level, b is a number of time unit in U from where the pattern begins, e is a number of time unit in U where the pattern ends, and p is a length of a period measured in the total number of time nits after the computations of T_s are repeated. The positional parameters b, c, p of a periodic pattern must satisfy the following properties: $1 \leq b < length(U)$ and $1 < e \leq length(U)$ and $\exists n \in 1, 2 \ldots e = b + n * p$.

Let A be an audit trail. A subsequence all SQL statements in an audit trail A processed in the n-th time unit in U is denoted by $A[n]$. Then, the total load created by the multiple processing of a statement s in $A[n]$ is denoted by $load(s, A[n])$. We say that a periodic pattern $<T_s, U, l, b, e, p>$ occurs in an audit trail A when $\forall l_s \in \{load(s, A[b]), load(s, A[b+p], \ldots, load(s, A[e])\} l_s \geq l$.

It may happen, that due to the random reasons, certain periodically repeated real world processes do not occur from time to time. Then a weaker definition of periodic patter is needed to describe such cases. Let $c = 1 + (e - b)/p$. We say that a periodic pattern $<T_s, U, l, b, e, p>$ occurs in an audit trail A with a support $0 < \sigma \leq 1$ when the total number of values in a set $\{load(s, A[b]), load(s, A[b + p], \ldots, load(s, A[e])\}$ that are greater or equal to a threshold load l is greater or equal to $\sigma * c$.

A process of discovering periodic patterns in an audit trails takes under the consideration a situation when some of the database applications are submitted for processing by different users with different frequencies. It means that the same statements can be involved in a number of periodic patterns with different frequencies. We also take under the consideration that two or more applications share and use the same module. Then, even if processing of some statements does not reveal any periodic patterns then it is possible that their common module may behave in a way consistent with a certain periodic pattern.

4.1 Reduced syntax tree table

Let $\mathcal{T}$ be a set of syntax trees that consists of all syntax trees of statements in an audit trail. Let T_ϵ be an *empty syntax tree* and let T_π be a syntax tree obtained from concatenation of all syntax trees from syntax tree table, which are not included in any other syntax tree. Then, discovering periodic patters in an audit trail is performed over a lattice $< \mathcal{T}, \sqsubseteq>$ implemented as a syntax tree table with a minimum T_ϵ and maximum T_π. The following three rules can be used to reduce the total number of iterations over the syntax trees.

(1) If a periodic pattern $<T_s, U, w, b, e, p>$ occurs in an audit trail A then for any syntax tree T such that $T \sqsubseteq T_s$ the same periodic pattern occurs in A.

(2) If a periodic patterns $<T_s, U, w, b, e, p>$ does not occur in an audit trail A then for any syntax tree T such that $T_s \sqsubseteq T$ the same pattern does not occur in A.

(3) If a periodic patterns $<T_s, U, w, b, e, p>$ does not occur in an audit trail A then for any syntax tree $T \sqsubseteq T_s$ and not shared with any other subtree the same pattern does not occur in A.

Copyright © 2013 SERSC

The rules listed above mean that for any syntax tree $T \sqsubseteq T_s$ and not shared with any other subtree a set of periodic pattern that occurs in T is the same as set of periodic patterns that occur in T_s. It allows to reduce a syntax tree table to a simple table of pairs $<tree, timestamps>$ where *tree* is an identifier of a syntax tree that suppose to be verified against periodic patterns and *timestamps* is a set of timestamps when the processing of a syntax tree identified by *tree* occurred in an audit trail. A *reduced syntax tree table* includes the identifiers of all sub-lattices determined by the rules (1)-(3) above. The table contains only information about the syntax trees of the statements from an audit trail and about subtrees shared by two or more syntax trees. For example, a syntax tree table given in Table 1 reduces to a set of pairs $\{<1, \{ts_1, ts_3, ts_4\}>, <2, \{ts_1\}>, <3, \{ts_2, ts_4\}>, <5, \{ts_4\}>\}$.

4.2 Workload histogram

Next, for each syntax tree T in a reduced syntax tree table a sequence of workload amounts W_T is created. A sequence W_T is called as *workload histogram* of a syntax tree T and it is used to represents the workloads imposed on a database system when processing a syntax tree T in each time unit in U. The n-th value in a workload histogram $W_T[n]$ is equal to $w_T * |T.timestamps[n]|$, where $T.timestamps[n]$ is a set of timestamps included in the n-th time unit and associated with the identifier of a syntax tree T in a reduced syntax tree table.

5 Discovering periodic patterns

Discovering a periodic pattern $<T, U, w, b, e, p>$ for a given set of time units U, a given minimal workload w, and a given value of support parameter $0 < \sigma \leq 1$ is performed through the nested iterations over the syntax trees included in a reduced syntax tree table and the iterations over the positional parameters b, e, and p. At the beginning all syntax trees in a reduced syntax tree table are marked as "not processed yet" and a set $\mathcal{P}$ of periodic patterns that occur in an audit trial A is set to empty. At each level the iterations are performed in the following way.

(1) At the outermost level we pick a syntax tree T from a reduced syntax tree table such that it is not included in any other "not processed yet" syntax tree. If such tree does not exist then the iterations are completed. Otherwise, we create a workload histogram W_T for T.

(1.1) At the first inner level the iterations are performed over the values of positional parameter b. The parameter b iterates over an increasing sequence of numbers $1, 2, 3, \ldots, |U| - 1$. Let b_c be the current value of parameter b. If $W_T[b_c] \leq w$ then a value of b_c is increased by one and the same condition is tested again. If no more iterations over the values of parameter b are possible then we move to a step (1.2) below.

(1.1.1) At the next inner level the iterations are performed over the values of parameter e for a fixed value b_c set at outer level. A parameter e iterates over a decreasing sequence of numbers $|U|, |U| - 1 \ldots, b_c + 2, b_c + 1$. Let e_c be the current value of parameter e. If $W_T[e_c] \leq w$ then we take the next value of parameter e the same condition is tested again. If no more iterations over the values of parameter e are possible we return to level(1.1).

 Copyright © 2013 SERSC

(1.1.1.1) At the lowest level the iterations are performed over an increasing sequence of values of parameter p such that $(e_c - b_c) mod\ p = 0$ and $b_c + p < e_c$. If no more iterations over the values of parameter p are possible we return to level (1.1.1). Otherwise, we set the current value of parameter p to p_c.

(1.1.1.2) Next, we create a candidate periodic pattern $<T, U, w, b_c, e_c, p_c>$ and we use a histogram W_T to check whether the candidate pattern is valid in an audit trail with a given support σ. Let W_T^+ be a set of all values of histogram W_T with the same numbers as a set of time units U_T and such each value in $W_T^+ \geq w$. Then, a candidate periodic pattern $<T, U, w, b_c, e_c, p_c>$ is valid in an audit trail with a support σ when $W[b_c] \geq w$ and $W[e_c] \geq w$ and $\sigma \leq |W_T^+|/|U_T|$.
If the candidate pattern is not valid in an audit trail then we return to step (1.1.1.1) to collect the next value of parameter p.

(1.1.1.3) If a candidate pattern is valid in an audit trail then we find the smallest workload in the pattern $w_{min} = min\{W_T[b_c + i * p_c], \forall i = 0, 1, 2, \ldots, (e_c - b_c)/p_c\}$ and we append $<T, U, w_{min}, b_c, e_c, p_c>$ to a set $\mathcal{P}$. Then we modify the entries of histogram W_T such that $W_T[b_c + i * p_c] := W_T[b_c + i * p_c] - w_{min}, \forall i = 0, 1, 2, \ldots, (e_c - b_c)/p_c$. Next we return to step (1.1.1.1) to collect the next value of parameter p.

(1.2) At the end of iterations over the positional parameters we are left with the single elements in a workload histogram W_T, which are not attached to any periodic pattern in $\mathcal{P}$ and such that their value is greater than w. If there exists a periodic pattern $< T, U, w, b, e, p > \in \mathcal{P}$ and an element $W_T[n] > w$ such that $n \in \{b+p, b+2*p, \ldots, e-2*p, e-p\}$ then we split the pattern into $< T, U, w, b, n, p >$ and $< T, U, w, n+p, e, p >$ and we modify a histogram $W_T[n] := W_T[n] - w$. Splitting the periodic patterns is repeated until no more single elements in W_T can be used.
When finished, we mark a syntax tree T as "processed" in a reduced syntax tree table and we return to a step (1) above.

6 Conclusions and further work

The efficiency of search over the dimensions of syntax trees and positional parameters of periodic pattern is very low. If an audit trail is divided by a set of time units U into n partitions then complexity of a search over the values of b, e, p is approximately $O(k * n^3)$ where $0 < k < 1/8$. Complexity of search over syntax trees is hard to estimate as it depends on the total number of access methods to relational tables, complexity of SQL statements, and a level of sharing common components among SQL statements.

As usual more efficient search over a space of syntax trees and positional parameters is a natural objective for the further research. As an ultimate objective is to apply the discovered periodic patterns to automated database performance tuning the next open problem is an application of the patterns to the prognostics of future intensity and structure of database workload. It requires a system of derivation rules for the periodic patterns to estimate what relational tables will be accessed by user applications and what database operations will be processed by the applications. One more issue is the right choice of a sequence of time units U when searching for periodic patterns. Too long time units in U would hide the existence of periodic patterns while too short time units would make the discovered patterns hard to comprehend and not consistent with the reality.

Copyright © 2013 SERSC

References

[1] Martín Abadi and Zohar Manna. Temporal logic programming. *J. Symb. Comput.*, 8(3):277–295, 1989.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of The 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 207–216, 1993.

[3] Marianne Baudinet, Jan Chomicki, and Pierre Wolper. Temporal deductive databases, 1992.

[4] Garrett Birkhoff. Lattice theory. In *Colloquium Publications*, volume 25. Amer. Math. Soc., 3. edition, 1967.

[5] Nicholas Bruno, editor. *Automated Physical Database Design and Tuning*. CRC Press Taylor and Francis Group, 2011.

[6] Jiawei Han, Wan Gong, and Yiwen Yin. Mining segment-wise periodic patterns in time-related databases. In *Proceeding of International Conference on Knowledge Discovery and Data Mining*, pages 214–218, 1998.

[7] Kuo-Yu Huang and Chia-Hui Chang. SMCA: A general model for mining asynchronous periodic patterns in temporal databases.

[8] Martin Krallinger, Alfonso Valencia, and Lunette Hirschman. Linking genes to literature:text mining, information extraction, and retrieval applications for biology. *Genome Biology*, 9(2), 2008.

[9] Srivatsan Laxman and P S Sastry. A survey of temporal data mining. *Sadhana, Academy Proceedings in Engineering Sciences*, 31(2):173–198, 2006.

[10] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1:259–289, 1997.

[11] Banu Özden, Sridhar Ramaswamy, and Abraham Silberschatz. Cyclic association rules. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pages 412–421, 1998.

[12] F. Rasheeed, M. Alshalalfa, and R Alhajj. Efficient periodicity mining in time series databases using suffix trees. *IEEE Transactions on Knowledge and Data Engineering*, 23(1):79–94, 2011.

[13] John F. Roddick and Myra Spiliopoulou. A survey of temporal knowledge discovery paradigms and methods. *IEEE Transactions on Knowledge and Data Engineering*, 14:750–767, 2002.

[14] Dan A. Simovici and Chabane Djeraba. *Mathematical tools for data mining : set theory, partial orders, combinatorics*. Advanced Information and Knowledge Processing. Springer, London, 2008.

[15] M. Wojciechowski. Discovering frequent episodes in sequences of complex events. In *Proceedings of Enlarged Fourth East-European Conference on Advances in Databases and Information Systems (ADBIS-DASFAA)*, pages 205–214, 2000.

[16] Jiong Yang, Wei Wang, and Philip S. Yu. Mining asynchronous periodic patterns in time series data. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):613–628, March 2003.

Copyright © 2013 SERSC

[17] Jieh-Shan Yeh, Szu-Chen Lin, and Shueh-Cheng Hu. Novel algorithms for asynchronous periodic pattern mining based on 2-d linked list. *International Journal of Database Theory and Application*, 5(4):33–43, 2012.

[18] Jieh-Shan Yeh, Szu-Chen Lin, and Shueh-Cheng Hu. OEOP: A novel algorithm for periodic pattern mining. *International Journal of Hybrid Information Technology*, 5(2):365–367, 2012.

[19] Marcin Zimniak, Janusz Getta, and Wolfgang Benn. Discovering periodic patterns in database audit trails. In *Proceedings of International Conference on Interdisciplinary Research Theory and Technology*, pages 365–367, 2013.

Copyright ⓒ 2013 SERSC