Control and Cybernetics

vol. **38** (2009) No. 1

Time complexity of page filling algorithms in Materialized Aggregate List (MAL) and MAL/TRIGG materialization cost*

by

Marcin Gorawski

Silesian University of Technology, Institute of Computer Science
Akademicka 16, 44-100 Gliwice, Poland
Marcin.Gorawski@polsl.pl

Abstract: The Materialized Aggregate List (MAL) enables effective storing and processing of long aggregates lists. The MAL structure contains an iterator table divided into pages that stores adequate number of aggregates. Time complexity of three algorithms was calculated and, in comparison with experimental results, the best configuration of MAL parameters (number of pages, single page size and number of database connections) was estimated. MAL can be also applied to every aggregation level in different indexing structures, like for instance the aR-tree.

Keywords: spatial data warehouse, materialization, indexing, materialized aggregate list, time complexity of MAL algorithms.

1. Introduction

In the relational data warehouse the query processing time can be decreased using adequate indexation and materialization methods. View materialization consists of first processing and then storing partial aggregates, which later allows the query evaluation cost to be minimized, performed with respect to a given load and disk space limitation (see Theodoratos and Bouzehoub, 2000). In Harinarayan, Rajaraman and Ullman (1996) the authors, for the first time, use the spatial network for storing the relations between aggregated views. In Baralis, Paraboschi and Teniente (1997), Gupta (1997), Gupta and Mumick (1999) views materialization is determined with workload and disk space limitation. Indices can be created on every materialized view, and in order to decrease the problem complexity, materialization and indexation are often used separately. This means that optimal indexation scheme is chosen for a given space limitation, just after the perspective set for materialization is defined (see

*Submitted: June 2008; Accepted: October 2008.

Golfarelli, Rizzi and Saltarelli, 2002). In Labio, Quass and Adelberg (1997) a set of heuristic criteria is proposed for choosing data warehouse views and indices. In Rizzi and Saltarelli (2003) authors present a comparative evaluation of benefits resulting from applying views materialization and data indexing in data warehouses, focusing on query properties. Next, an heuristic evaluation method was proposed for a given workload and global disk space limitation.

The reminder of the paper is organized as follows: Section 2 briefly describes the motivation for our work. In Section 3 the MAL architecture along with new definitions for the MAL iterator are presented. Section 4 describes the details of the proposed page-filling algorithms for the MAL iterator and, presented for the first time, formulas for their time complexity. In this section we compare the theoretical MAL algorithm graphs with the results obtained earlier (Gorawski, Malczok, 2005). Finally, Section 5 concludes the paper.

2. Motivation

We are working in the field of spatial data warehousing. Our system (Distributed Spatial Data Warehouse, DSDW), presented in Gorawski and Malczok (2004), is a data warehouse gathering and processing huge amounts of telemetric information generated by the telemetric system of integrated meter readings. The readings of water, gas and energy meters are sent via radio through the collection nodes to the telemetric server. A single reading (measurement) sent from a meter to the server consists of: precise timestamp, meter ID and a measurement value. Periodically, the extraction system loads the data to the database of the data warehouse system. In our current research we are trying to find the weakest spots in the current system. After various test series (with variations of aggregation periods, numbers of telemetric objects etc.) we found that the crucial problem is to create and manage a long aggregate list. The aggregate list is a list of meter readings, aggregated according to an appropriate time window. A time window is a time period in which we want to investigate utility consumption. The aggregate consists of a timestamp and aggregated values. When there is a need of investigating the utility consumption, we have to analyze consumption history. This is when the aggregate list comes useful. Every aggregate belonging to the list, stores several values, so the memory usage is rather high. To prevent the memory overflow problem, appropriate memory managing algorithm was designed, presented in Gorawski and Malczok (2004). However, even if aggregate value calculation is efficient in simple scenarios like summing up several values, time needed to calculate a single aggregate, multiplied by the length of aggregates list results in a significant prolonging of the aggregate values calculation process. This is why in many cases the decision is made to store some or all of the already calculated aggregates - this process is called materialization. We propose a new method for storing and materializing aggregates. For this method of aggregate materialization we made the following assumptions:

1. materialization eliminates memory restrictions, and
2. enables storing of any type of attributes and allows for various types of data sources, and
3. bases on a well-known program structure.

The method is called the Materialized Aggregate List (MAL) (Gorawski and Malczok, 2005).

3. Architecture of Materialized Aggregate List

The Materialized Aggregate List (MAL) was designed based on a scheme of a list available in Java language (*java.util.List* (http://java.sun.com)). Fig. 1 presents the schema of the Materialized Aggregate List architecture.

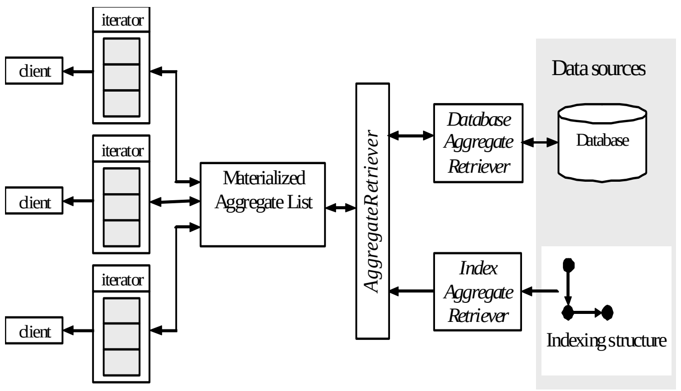

Figure 1: Schema of the MAL architecture.

In comparison to *java.util.List*, MAL architecture is more complex. The first significant difference is data source of a list. MAL elements are loaded directly from a data source (defined for a specific list and instance), not through its clients. For a Materialized Aggregate List client (MAL client) the type of list data source is transparent. The second aspect that strongly distinguishes the materialized list from the ordinary list is the shift of functionality to iterators. Clients can communicate with MAL only using iterators. Iterators have quite complex structure, ensuring adequate cache and aggregate materialization mechanisms. The last difference worth mentioning is the fact that the list object stores no data (aggregates). Its role is limited to being a mediator between iterators used by clients and a data source. The MAL client browses the list, retrieves aggregates and processes them in a characteristic manner.

3.1. The MAL iterator

Iterators of Materialized Aggregate List (MAL iterators) allow only browsing the list and retrieving aggregates stored in the list. According to the concept of DSDW the delete operation is unavailable. Browsing and list element retrieval is realized with two functions of the iterator:

1. $hasNext()$ - the method returns a boolean type logical value. The function returns value $true$ if the list contains next element, or $false$ if it does not contain next elements.
2. $next()$ - the method enables retrieving a current list element and automatically moves iterator pointer (cursor) to the next list element.

In comparison with standard list iterator, the MAL iterator architecture is much more complex. Every iterator contains memory structure - a table of strictly defined size.

Definition 1 *The table of the MAL B iterator (B) is a memory structure that stores aggregates. The B table is a set of logical fragments called pages.*
$B := \{S_i : S_i$ *- i-th page of a MAL iterator,* $i \in 1, \ldots, n\}$*, where* n *- number of pages.*

Pages store a fixed number of aggregates. For every page the number of aggregates is constant. Those logical pages find their usage during the aggregate creation process. The value that describes a page of a table of MAL iterator is a timestamp of the first aggregate, called *border date*. Fig. 2 presents the schema of the Materialized Aggregate List iterator architecture.

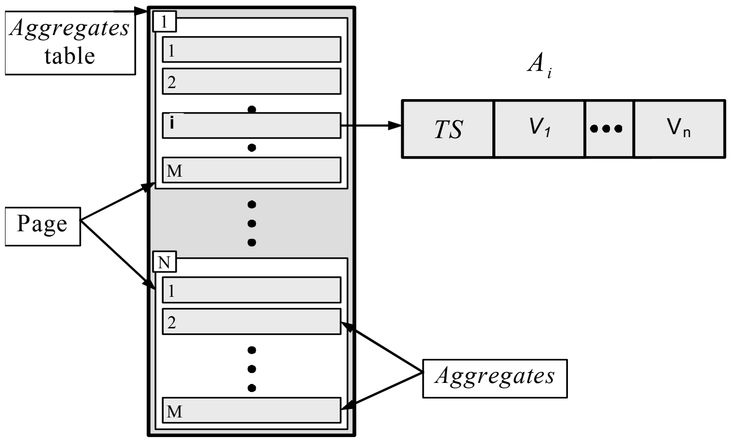

Figure 2: Schema of a MAL iterator.

Definition 2 *Page S of MAL iterator table (S) is a logical fragment of the B table. Page S is a set of aggregates:*
$S := \{A_i : A_i$*– i-th aggregate,* $i \in 1, \ldots, M\}$*, where M – number of aggregates. Page S is defined with border date BD, which is equal to the timestamp of the first aggregator in the page, decreased by this aggregate time window value.*

The smallest structure stored in the table is an aggregate. The aggregate consists of a timestamp and a value set. The aggregate values have certain types (e.g. a floating point number, an integer). Aggregates are calculated for a certain time span, called aggregate time window.

Definition 3 *An aggregate A is the smallest structure stored in the MAL iterator table. Aggregate A is defined as:* $A_i = (TS, V_{Ai})$ *where:*

1. *TS is the timestamp of the aggregate,*
2. *V_{Ai} is the list of V_{Ai} aggregate element values such as: $V_{Ai} := \{V_i : V_i$ - value of i-th V_i aggregate element, $i \in 1, \ldots, n\}$ and $\forall i \forall V_i \exists! t_{v_i}$, where t_{v_i} is aggregate type.*

The cardinality of a collection V_a ($|V_a|$) is not limited and is defined by the need of using a definite aggregate. The type of an aggregate is defined by the number of elements in the V_a collection and the types of elements in the collection. The list of values contains pre-processed data for aggregate calculation (e.g. the utility consumption).

Definition 4 *Aggregate type equality.Two aggregates A_m, A_n are of the same type if:*

1. *Cardinality of an A_m value list equals the cardinality of A_n value list, $|V_{A_m}| = |V_{A_n}|$,*
2. *Values of A_m and A_n located under the same indices are of equal simple types $i = j \Rightarrow t_{V_i} = t_{V_j}; i, j = 1, ..., |V_{A_m}|$.*

The MAL can be applied as a component of nodes in a hierarchical indexing structure. In such a case, the data source for higher level nodes of the structure is stored on lower levels. Aggregates in lists on higher levels are created by summation of the aggregates from lists stored on lower levels. We can sum only aggregates with identical timestamps and types.

Definition 5 *Aggregate addition, $A_s = A_m + A_n$. Two aggregates can be added if:*

1. *They have equal timestamps: $TS_{A_m} = TS_{A_n}$,*
2. *They are of equal type according to Definition 1.*

The resulting aggregate A_s has:

1. *a timestamp TS equal to timestamps of aggregates A_m and A_n: $TS_{A_s} = TS_{A_m} = TS_{A_n}$,*
2. *V_a being the list of aggregate A_s values. The elements of V_{A_s} are sums of elements of V_{A_m} and V_{A_n} : $V_i \in V_{A_s}, V_i = W_i + Q_i, i = 1|V_{A_s}|$, where: $W_i \in V_{A_m}$ and $Q_i \in V_{A_n}$.*

4. The MAL page filling algorithms and their time complexity

The MAL iterator table includes aggregates that can be browsed and retrieved by the MAL clients. The process of iterator table filling is executed using specially designed algorithms. Those MAL algorithms operate basing on the concept of a page as a logical part of the table. By the analysis of the table fulfillment status and actual cursor position that is set by client (with iterator $next()$ function), the MAL algorithm triggers a process of filling the appropriate page of the MAL iterator page.

Operation of a MAL client can be described by a single T_a aggregate consumption (processing) time. This time depends on an analysis specific for client operation. The analysis is conducted with regard to the page filling algorithm choice and configuration, and it is easier with definition of two temporal values:

1. T_k - the consumption time of a single page that describes client process; the T_k time is directly proportional to consumption time of a T_a aggregate, and the proportionality factor is the page size: $T_k = |S|T_a$, where $|S|$ is the page size. If a page stores aggregates from one day and aggregates are stored every 30 minutes, the page size equals 48.
2. T_w - the page filling time; page filling time directly depends on the used page filling algorithm. Factors that have the biggest influence on T_w value are presented below.

Our idea of MAL usage as a substitute for a memory list causes that client functioning cannot be put on hold while browsing the list. Operation of the client browsing the list can be halted only when a requested page does not contain any aggregates, because they have not been calculated. Basing on the MAL iterator idea and on definitions, which determine its structure, we can specify when the client functioning will not be halted. In the MAL we assume that every page filling is performed in a separate thread, which is independent of the list client thread. Different approaches to the page filling problem in a MAL iterator table are called *MAL page filling algorithms* or *MAL algorithms.* Below we present three such algorithms. The MAL algorithm descriptions have some parts in common and every one of them is characterized by:

1. The algorithm overhead, denoted as N. Before its actual functioning, the algorithm fills a definite number of pages. This action protects the client from a deadlock, in the case when the T_k time is too short. The unit of an overhead is a table page of a MAL iterator.
2. The boundary ratio of a T_w page filling and a page consumption time T_k, $R = \frac{T_w}{T_k}$, above which the considered page filling algorithm is unable to assure continuity of a MAL client functioning.

The $T(n)$ time complexities of MAL algorithms, where n denotes the number of aggregates retrieved from an iterator table, are quite similar, the differences result only from the overabundance factor N, that is different for every algorithm. As a dominant operation in terms of time we can assume the data source access operations and aggregate calculation process. The time complexity of a MAL algorithm is directly proportional to the number of pages that were filled with aggregates. The $S(n)$ memory complexity of those algorithms is identical for all versions. The quantity of memory (space) used to deliver aggregates to MAL client is strictly dependent on iterator list size, iterator table parameters and parameters of iterator table set.

4.1. The SPARE algorithm

Two first pages of the table are filled when a new iterator is being created and the SPARE algorithm is used as a page-filling algorithm. Then, during the list browsing, the algorithm checks if the last aggregate is retrieved from $n \cdot mod|B|$ page. If so, the $(n+2)mod|B|$ page is filled, while the client retrieves aggregates from the $(n+1)mod|B|$ page. One page is always kept as a "reserve", being a *spare* page. Algorithm usage is presented in Fig. 3. The SPARE algorithm characteristics are:

1. The algorithm overhead is one page: $N_{SPARE} = 1$
2. The time complexity $T(n)$ is calculated for two cases, assuming that:

Assumption 1 *The SPARE algorithm assures continuous functioning of a MAL client, if the page filling time is less or equal to the page consumption time* $T_w \leq T_k$. *So, the boundary value is:* $R = \frac{T_w}{T_k} = 1$.

1. There are only two pages of a MAL iterator table available. The pages are used as follows: one page is accessed at a moment (aggregate processing), the second one is reserved for readout when the first page processing finishes. The third page is needed to start filling it in a moment when the last aggregate is retrieved from the first page.
   With such assumptions we can present the time complexity of the SPARE algorithm for the case when there are only two pages of MAL iterator table available:

   $$T(n) = (2+p)T_w + pT_a \tag{1}$$

   where

   - T_w - the page filling time;
   - T_a -the consumption time of a one aggregate;
   - p - the number of pages read.

   The T_a time results from the fact that when the last aggregate from the first page is retrieved, the SPARE algorithm can not begin to fill the next page, because no page is yet available. The algorithm has to wait for page to be released by a thread that consumes aggregates - the waiting time is the time T_a needed for consumption of the last aggregate.

2. There are three or more MAL iterator table pages available. In this case the time complexity of the SPARE algorithm is

$$T_n = 2T_w + pT_w = (2 + p)T_w. \tag{2}$$

Greater number of available pages (more than three), from theoretical point of view, has no effect on efficiency of this algorithm. The algorithm does not assume usage of the greater number of pages.

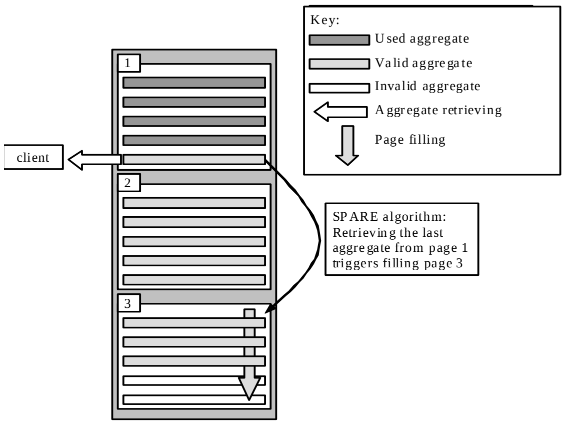

Figure 3: Schema of the SPARE page filling algorithm operation.

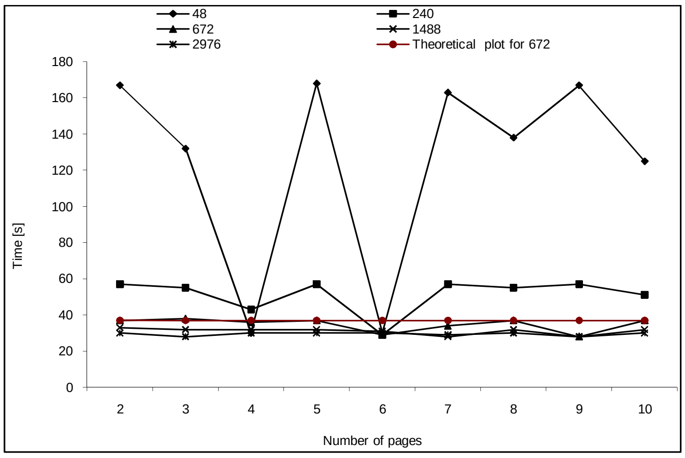

Figure 4: Time of task execution using the SPARE algorithm as a function of the number of pages and page size.

The experimental results (see Gorawski and Malczok, 2005) show that the best times were obtained for a larger number of pages than it would seem from theoretical analysis (Fig. 4.). This fact results from the thread synchronization mechanism that cannot free the read pages fast enough. For an additional page, the thread that fills a page does not wait for the page to be made accessible. However, use of additional pages generates memory management costs, which can be observed in Fig. 6 and decreases the overall efficiency of the SPARE algorithm.

4.2. The TRIGG algorithm

The TRIGG algorithm fills only one page of a MAL iterator table before it starts working. While browsing aggregates from the $n \cdot mod|B|$ page, the algorithm checks if the second to last aggregate was retrieved. The fact of retrieving the second to last aggregate from the $n \cdot mod|B|$ page is a *trigger* for a thread that fills the $(n+1)mod|B|$ page. The TRIGG algorithm does not fill additional pages until MAL client requests it (Fig. 5).

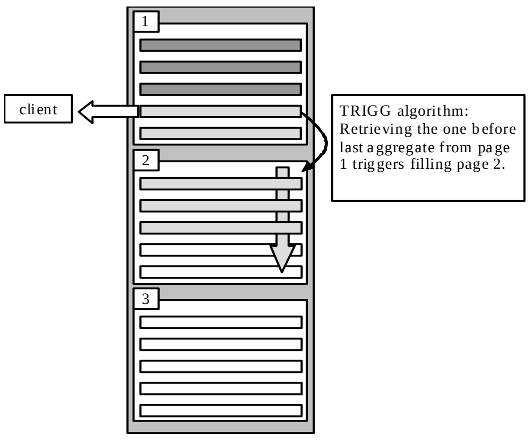
1
2
3
client
TRIGG algorithm: Retrieving the one before last aggregate from page 1 triggers filling page 2.

Figure 5: Functioning of a TRIGG page filling algorithm.

The features of the TRIGG algorithm are as follows:

1. The TRIGG algorithm has no overhead, $N_{TRIGG} = 0$.
2. The complexity of the TRIGG algorithm is: $T_n = pT_w$
3. The TRIGG algorithm assures fluent functioning of a MAL client only if the page consumption time is significantly greater than its filling time. So the boundary value is: $R = \frac{T_w}{T_k} = \frac{1}{|S|}$.

When determining the time complexity of a TRIGG algorithm, the dominant operation is retrieving data from a database (Fig. 6). The search for adequate

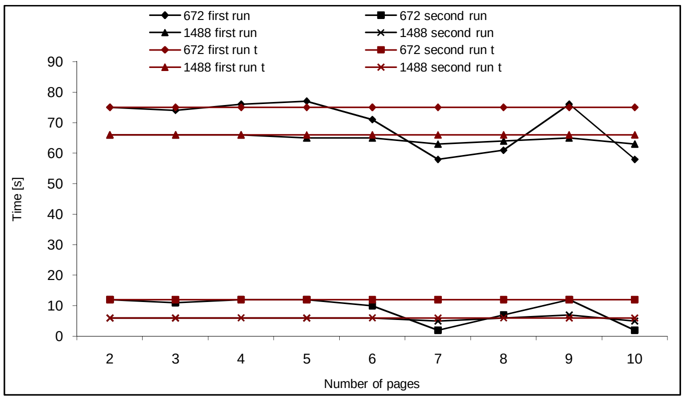

Figure 6: Positive influence of materialization on system operation.

data in a measurement table has the greatest influence on data retrieval. However, the materialized data are easily accessible and their number is significantly lower than of raw data, so access to such data is more cost efficient. As an example, the materialized data for one counter and one year will be stored in only 365 rows, not in 17520. So, the acceleration factor can be defined as a quotient of time of materialized data retrieval and time of raw data retrieval. For the TRIGG algorithm it is as follows:

$$T_n = \frac{T_m}{T_n}(pT_w) \tag{3}$$

where

- T_w – page filling time;
- T_m – materialized data retrieval time;
- T_n – raw data retrieval time.

4.3. The RENEW algorithm

The RENEW algorithm fills all pages of a MAL iterator table before it starts working. Then, when aggregates are browsed and retrieved by the client, the algorithm checks if the last aggregate was retrieved from page $n \cdot mod|B|$. If the client retrieved the last aggregate from $n \cdot mod|B|$ page and proceeded to $(n+1)mod|B|$ page, the algorithm starts a thread that fills $n \cdot mod|B|$ page with aggregates – the page is *renewed* to be browsed by the client process.

The features of the RENEW algorithm are as follows:

1. The RENEW algorithm has the highest overhead of the presented MAL algorithms – the overhead depends on the number of pages in iterator table and equals $N_{RENEW} = |B| - 1$.

2. Time complexity is also the highest:

$$T_n = (|B| - 1)T_w + pT_w = (|B| + p - 1)T_w. \tag{4}$$

3. To define the values of the R factor we have to consider additional conditions.

Each page of the MAL iterator table is filled using an independent thread. In the case of two previous algorithms, this fact had no influence, however, in the RENEW algorithm to precisely define conditions determining appropriate usage of the algorithm there is a need to consider additional Assumption 2.

ASSUMPTION 2 *The time T_w of page filling is independent of the number of simultaneously functioning threads.*

In the first approach to this problem we assumed that the system, where MAL is implemented, has enough resources to start any number of parallel threads that fill table pages and also that T_w time value for every thread will not be prolonged. In this case the T_w time has to be lower or equal to the page consumption time T_k and the number of pages minus one: $T_w \leq (|B| - 1)T_k$. The boundary value of the R ratio of times, in which list client functioning will not be halted, equals : $R = \frac{T_w}{T_k} = |B| - 1$. Further problem analysis shows that Assumption 2 can not be satisfied. So, the assumption has to be modified.

ASSUMPTION 3 *The time of page filling depends on the number of simultaneously functioning threads.*

The change of assumption implies defining a new time value – an effective time of page filling operation T_{we}. This time is directly proportional to the number of simultaneously functioning page filling threads (minus the thread, for which time is calculated). The value of T_{we} time equals:

$$T_{we} = (1 + k \times w)T_w \tag{5}$$

where:

- k - factor of mutual influence of threads;
- w - number of simultaneously functioning threads;
- T_w - page filling time.

The k factor determines the influence of other threads on the current thread, for which the T_{we} value is calculated; the factor has values $k \in [0, 1]$. The value of k enables to adequately modify formula (5), according to the capacity of a definite system. To assure the continuous functioning of a MAL client, it is obligatory to fulfill the condition: $T_{we} \leq (|B| - 1)T_k$. The value of R that determines the boundary value of a quotient of page filling time to page consumption time for the RENEW algorithm equals $R = \frac{(|B|-1)}{(1+kw)}$.

4.4. The influence of the number of database connections on MAL algorithms

The TRIGG algorithm utilizes only one database connection, so its time complexity is:

$$T_n = pT_w. \tag{6}$$

The second algorithm, RENEW, can utilize all of the pages of MAL iterator, so the T_n value equals:

$$T_n = \begin{cases} \frac{(|B|=p-1)(1+k(w-1))T_w}{|C|} & \text{for } |C| \leq |B| \\ \frac{(|B|+p-1)(1+k(w-1))T_w}{|B|} & \text{for } |C| > |B| \end{cases}. \tag{7}$$

While creating graph of Fig. 7 for this algorithm we assumed that k = 0.85, because this value adequately shows the mutual influence of threads in a single-processor system.

Time complexity of the SPARE algorithm T_w equals:

$$T_n = \begin{cases} \frac{(2+p)(1+k(w-1))T_w}{|C|} & \text{for } |C| < 3 \\ \frac{(2+p)(1+k)T_w}{2} & \text{for } |C| \geq 3 \end{cases}. \tag{8}$$

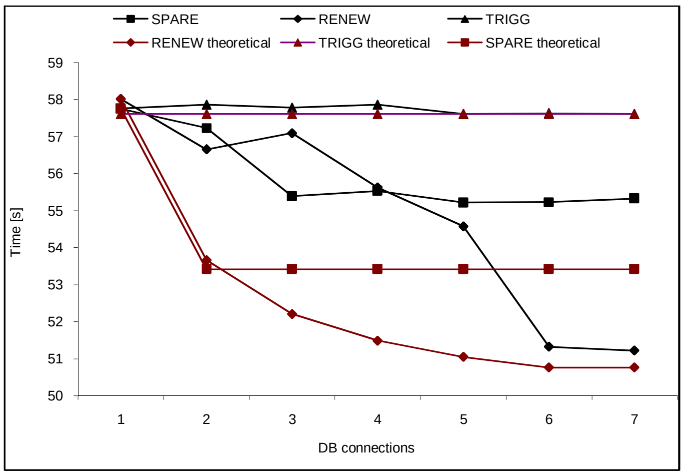

Figure 7: The influence of the number of database connections on MAL algorithms.

The results of experiments obtained in Gorawski and Malczok (2005) are consistent with theoretical assumptions (Fig. 7). Different measurement values in the function of the number of available connections for the TRIGG algorithm are within the boundaries of the measurement error. The results for the SPARE algorithm suggests that the MAL instance utilizes only one connection available in a connection pool. Because the time complexity formula (8) does not take into consideration this influence, there are some differences in the graph for this algorithm. A graph for the RENEW algorithm shows that there is also only one connection utilized by a MAL instance. The real (see Gorawski and Malczok, 2005) and theoretical characteristics for the RENEW algorithm differ, however, in the final point they become similar. The differences stem from a random behavior of the Java environment (removal of unused objects in random moments), thread waiting-time, imperfect synchronization.

5. The MAL/TRIGG materialization costs

One of MAL features is that once calculated and loaded to memory, aggregates are not lost, but they are stored in a table specially prepared for this purpose. This enables minimization of a page filling cost. Below, basing on the example of the TRIGG algorithm, we present the analysis of the materialization influence on data retrieving and processing. The following assumptions were considered: page size equal 672 (the best value obtained in tests presented in Gorawski and Malczok, 2004), aggregates will be retrieved for one counter for a time period of 12 moths, a single aggregate has only one float value, and the database is Oracle 10g.

Using (6) we can define time complexity of the TRIGG algorithm, when there are materialized data available and when there are no materialized data available. It is worth mentioning that the number of accesses to database for the materialized data is $|S|$ times less than the number of accesses to database when querying the fact table; $|S|$ is a single page size. The dominant operation while page filling is data retrieval from a database so we can replace the T_w factor with the T_m and T_{nm} parameters denoting data retrieval cost.

The value of $T(n)$ for the immaterialized (materialized) data is given by (9) (or (10)):

$$T(n_{nm}) = pT_{nm} = p(|S|DBC_{nm} + PC) \tag{9}$$

$$T(n_m) = pT_m = p(|S|DBC_m + PC) \tag{10}$$

where:

- $T(n)$ – materialized data retrieval time;
- $T(nm)$ – immaterialized data retrieval time;
- p – number of processed pages;
- $|S|$ – single page size;
- PC – cost of a page creation from the application side;
- DBC_m – query execution cost for materialized data;
- DBC_{nm} – query execution cost for immaterialized data.

After several transformations we can present $T(n_m)$ using $T(n_{nm})$

$$T(n_m) = pT_m l_z \frac{T(n_{nm})}{T(n_{nm})}) = \frac{T(n_m)}{T(n_{nm})}(pT(n_{nm})) = \frac{T(n_m)}{T(n_{nm})}T(n_{nm}). \quad (11)$$

So, the acceleration factor can be defined as a quotient of materialized data retrieval time and raw data retrieval time:

$$T(n_m) = \frac{T(n_m)}{T(n_{nm})}(pT(n_w)) \quad (12)$$

where:
- T_w – page filling time;
- T_m – materialized data retrieval time;
- T_{nm} – immaterialized data retrieval time;
- p – number of processed pages.

The factor pT_w denotes task competition time, when no materialized data is available. It can be presented without accuracy loss as pT_{nm}, however, because of the earlier page filling the notation pT_w and the former dependence are chosen. To determine pT_{nm} and pT_m, we have to calculate the cost of materialized and immaterialized data access. The materialized data table does not contain any indices, which accelerates record search, so we have a full table scan (Table Access (FULL)). The cost of such operation, presented with an I/O access count is estimated as (see Wojciechowski and Zakrzewicz, 2002):

$$cost = \left[K_m \frac{blocks + 1}{db\_file\_multiblock\_read\_coun} \right] \quad (13)$$

where:
- *blocks* – number of table blocks; by increasing this value by one we account for one header block of a table segment;
- *db_file_multiblock_read_count* – system parameter that defines the size of multiblock sequential readouts, number of blocks retrieved in a single disc access;
- K_m – correction factor, depending on the *db_file_multiblock_read_count* value (Table 1), it represents the risk of a sequential readout prolongation in the case of hitting the end of disc cylinder or the end of the extent; this risk depends on the readout size;

The database table that stores materialized pages of an iterator table consists of three columns. The first column stores the identifier of the object that materialized data considers. The second column stores page boundary dates and the third column stores materialized data in a binary format. The example of such table is presented in Fig. 8a and a structure of a BLOB column is presented in Fig. 8b.

Table 1: The K_m correction factor values for the sequential readouts.

db_file_multiblock_read_count	K_m
1	0.596500
2	0.755967
3	0.868342
4	0.958064
5	1.033995
6	1.100481
7	1.160019
8	1.214190
9	1.264068
10	1.310419

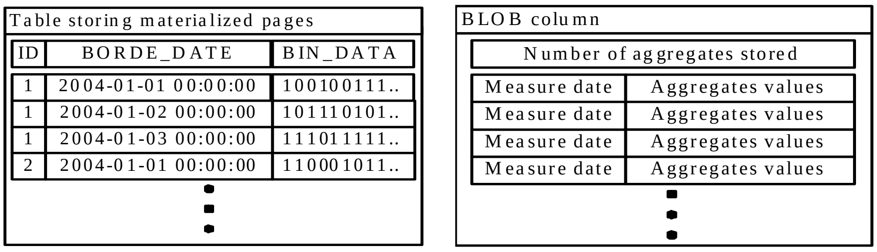

Table storing materialized pages

ID	BORDE_DATE	BIN_DATA
1	2004-01-01 00:00:00	100100111..
1	2004-01-02 00:00:00	101110101..
1	2004-01-03 00:00:00	111011111..
2	2004-01-01 00:00:00	110001011..

BLOB column

Number of aggregates stored

Measure date	Aggregates values
Measure date	Aggregates values
Measure date	Aggregates values
Measure date	Aggregates values

(a) (b)

Figure 8: (a) The table storing materialized aggregates; (b) The BLOB column structure.

Measure_date (measurement date) is always created with 6 int-type numbers. The information on the number of stored aggregates fits in a single int-type number. The size of Aggregate values field is dependent of MAL settings and this information can be obtained in DWE application. The single aggregated page size equals

$$MPS = n * (date\_size + agg\_size) + 4 \tag{14}$$

where:

- MPS – the size of single materialized page given in bytes
- n – the number of aggregates on a page
- $date\_size$ – the size of a single timestamp; now it equals : 6*4 = 24 bytes
- agg_size – the size of information about value(s) of a single aggregate, depending on the MAL settings.

Number 4 was added to include the size of a field that contains information about the number of stored aggregates on a materialized page. This number does not have to be equal to the MAL page size, because there is not always adequate number of aggregates available. As it was stated earlier, the page size is 672 and a single aggregate contains only one float value (size - 4 bytes), so the size of a materialized page is:

MPS=672*(24+4)+4=18820 B.

If we assume that a single block has 8192 B (standard Oracle setting), and one year consists of 17856 readouts (12 months * 31 days * 48 readouts), then the number of blocks filled with data equals:

- Page size - 672($14 days$)
- Row size [B] - 18820
- Number of rows for a single year - $\frac{17856}{672} = 27$
- Number of block needed to store one year's data -$\frac{27*18820}{8192} = 63$.

Assuming that the $db\_file\_multiblock\_read\_count$ factor value equals 8 (default value at database creation), then a cost of a single materialized page retrieval is, see (13):

$$T_m = \left\lceil K_m \frac{MaterializedBlocks + 1}{cd\_file\_multiblock\_read\_count} \right\rceil = \left\lceil 1,21419 * \frac{64}{8} = 10 \right\rceil . \quad (15)$$

Query retrieving data from a fact table proceeds as follows:

```
      SELECT max(value) - min(value) as value FROM measure m
        WHERE mts > start_date AND mts < end_date
          AND meter_id = id
    GROUP BY zone
    ORDER BY zone
```

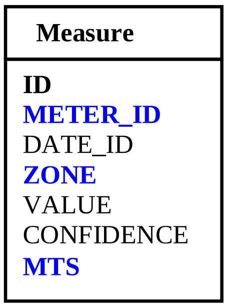

Figure 9: The MEASURE table structure.

The schema of the fact table (MEASURE) is presented in Fig. 9. In this table we created database index for *meter_id*, *zone* and *mts* columns. It is a B*-tree type index.

The cost of an access to this table for the above-mentioned query equals 5. This value is a sum of the following elements:

- the readout of a header of an index segment (cost = 1),
- the readout of an index root (cost = 1),
- the readout of an index leaves percentage, equal to a selection condition selectivity ($mts > start\_ date\ AND\ mts < end\_ date\ AND\ id = meter\_ id$); it depends on the number of readouts in a certain time period. It was assumed that there are two readouts and they both are stored in the same block (cost = 1),
- table blocks readout, pointed with index leaves pointers; only one block is accessed (cost = 1),
- sorting of the obtained results (cost = 1).

To fill a singe page, there has to be a number of accesses to the database available, which equals the size of an iterator table, so the cost of single page filling with raw data is:

$$T_{nm} = |S| DBC\_B * tree = 672 * 5 = 3360 \tag{16}$$

where:
- $|S|$ – number of aggregates on a page; page size
- $B * tree$ – cost of a database access with a B*-tree usage.

By putting together (12) with (15) and (16) we get the value of acceleration caused by materialization:

$$T_n = \frac{T_n}{T_{nm}} pT_w = \frac{10}{3360} pT_w = 0,003pT_w \tag{17}$$

where:
- T_w – page filling time
- T_m – materialized data retrieval time
- T_{nm} – immaterialized data retrieval time
- p – number of processed pages.

On the basis of the above presented calculations we can say that costs of a database access for the materialized data are minimal. However, with the increasing volume of materialized data the costs of retrieval will increase linearly, and the cost of a fact table data retrieval, with usage of a B*-tree type index, will increase logarithmically. That is why we have to assume that in the case of whole fact table materialization (of a much greater size than presented in this paper), the materialization benefits will decrease and in some pessimistic cases the whole cost will even increase.

5.1. Optimization

It seems that additional materialization benefits can be obtained by decreasing the data size that is stored in a BLOB column. The timestamp can be written not in 6 integer type numbers but in a single long type number that will store the number of milliseconds passed since 1970. It is connected with additional overhead while processing a date stored in such a manner, however, arithmetical operations are far less time consuming than data retrieval from a database. The other manner of date storing can be coding the whole date in a bit form. In such a case only one integer type number is sufficient: 6 bits - minutes, 5 bits - day, 4 bits - month, 12 bits - year. The sum is 32 bits so the same as an integer type number. With simple bitwise operations, adequate field can be identified. Unfortunately, such approach will cause the code to be less legible. Moreover, there is a possibility of materialized data size limitation, resulting from the MAL characteristics. The aggregates defining utility consumption are calculated for a certain time window, whose width is defined in a configuration file, separately for each counter type. The timestamp of a next aggregate equals the timestamp of previous aggregate + the time window width. So, the timestamp of an n-th measurement is

$$TS_n = BD + nTW \tag{18}$$

where:

- TS_n - the n-th aggregate timestamp
- BD - the page border date
- TW - the time window.

Using the MAL properties, there is a possibility of a complete removal of a timestamp in a BLOB column of a materialized aggregates table. As a replacement it is enough to store the size of a time window in the following manner: unit type and a number of units. Both these pieces of information can be stored in two single integer type numbers. It will cause a great decrease of volume of stored information and simultaneously the cost of a next timestamp calculation to be negligible, with insignificant overhead. The page border date is known on the level of selection condition formulation for the next page of a materialized list, hence there is no need to retrieve this information from a data base. The size of such "clipped" record in a materialized data table, considering earlier assumptions (672 aggregates, single float type value) equals:

$$MST = (673 * 4) + 12 = 2700B. \tag{19}$$

Number 12 stands for three int type numbers storing information about number of aggregates on a page, the time windows time units and the number of those units. As a result the size of materialized data decreased more than 6 times.

6. Conclusion

Three versions of a MAL page filling algorithm were presented. For a MAL client those algorithms vary mainly in overhead and possibility of a fluent aggregate delivery for different page consumption times. The first and most important factor of an algorithm selection is the time of aggregate processing by a MAL client process. In the case when the consumption time of a MAL iterator table page is comparable with the page filling time, the SPARE algorithm that can assure continuous client functioning with insignificant overhead is the best choice. However, if the consumption times are small, or aggregate calculation is time consuming, then RENEW algorithm should be used. Another feature that determines the adequate algorithm selection is the character of the system, in which MAL is implemented. If the system resources (mainly the available throughput of I/O channels) are rather small, then the RENEW algorithm will not guarantee fluent functioning of a MAL client. On the example of the TRIGG algorithm we presented the analysis that confirms favorable influence of materialization on data retrieval and processing. It was assumed that in case of whole fact table materialization, the materialization benefits will decrease and in some pessimistic cases the whole cost will even increase (research still in progress). On a current designing stage the page filling algorithms as well as other MAL parameters, are defined in an XML configuration file, so they are not dynamically balanced with a DSDW load. The other manner of a MAL functioning speed-up is to use the LRU buffer as a cache mechanism of a higher hierarchical level.

References

Baralis, E., Paraboschi,, S. and Teniente, E. (1997) Materialized view selection in multidimensional database. In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases.* Morgan Kaufmann, 156-165.

Golfarelli M., Rizzi S. and Saltarelli E. (2002) Index selection for data warehousing. In: *Proceedings of 4th International Workshop on Design and Management of Data Warehouses, DMDW'2002*, Toronto, Canada. CEUR-WS **58**, 33-42.

Gorawski M. and Malczok R.(2004) Distributed Spatial Data Warehouse Indexed with Virtual Memory Aggregation Tree. *Proceedings of 2nd International Workshop on Spatio-Temporal Database Management, STDBM'04*, Toronto, Canada. Morgan Kaufmann, 25-32.

Gorawski M. and Malczok R.(2005) On Efficient Storing and Processing of Long Aggregate Lists. *7th International Conference Data Warehousing and Knowledge Discovery, DaWaK.* **LNCS 3589**, Springer Verlag, 190-199.

Gupta, H. (1997) Selection of views to materialize in a data warehouse. *Proceedings of 6th International Conference ICDT'97, Delphi, Greece.* **LCNS 1186**, Springer Verlag, 98-112.

Gupta, H.and Mumick, I.S.(1999) Selection of views to materialize under a maintenance cost constraint. *Proceedings of 7th International Conference ICDT'99*, Jerusalem, Israel. **LCNS 1540**, Springer Verlag, 453-470.

Harinarayan, V., Rajaraman, A. and Ullman, J. (1996) Implementing data cubes efficiently. *ACM SIGMOD Rec.* **25** (2), 205-216.

Labio W.J., Quass D. and Adelberg B. (1997) Physical database design for data warehouses. *Proceedings of 13th International Conference on Data Engineering*, Birmingham, UK. IEEE Computer Society, 277-288.

Rizzi S. and Saltarelli E.(2003) View Materialization vs. Indexing: Balancing Space Constraints in Data Warehouse Design. *Proceedings of 15th International Conference on Advanced Information Systems Engineering*, CAiSE 2003, Klagenfurt, Austria. **LCNS 2681**, Springer Verlag, 502-519.

Sun Microsystems Sun Microsystems JavaTM 2 Platform Standard Edition 5.0 API Specification. *http://java.sun.com.*

Theodoratos, D. and Bouzehoub, M. (2000) A general framework for the view selection problem for data warehouse design and evolution. *Proceedings of the 3rd ACM international workshop on Data warehousing and OLAP*, McLean, Virginia, US. ACM Press, 1-8.

Wojciechowski M. and Zakrzewicz M. (2002) Cost-based query optimizer *III PLOUG Oracle Seminar.* Polish Oracle User Group, Warszawa, 5-16.