On Efficient Storing and Processing of Long Aggregate Lists

Marcin Gorawski and Rafal Malczok

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

Abstract. In this paper we present a solution called Materialized Aggregate List designed for the efficient storing and processing of long aggregate lists. An aggregate list contains aggregates, calculated from the data stored in the database. In our approach, once created, the aggregates are materialized for further use. The list structure contains a table divided into pages. We present three different page-filling algorithms used when the list is browsed. We present test results and we use them for estimating the best combination of the configuration parameters: number of pages, size of a single page and number of available database connections. The Materialized Aggregate List can be applied on every aggregation level in various indexing structures, such as, an aR-tree.

1 Introduction

Query evaluation time in relational data warehouse implementations can be improved by applying proper indexing and materialization techniques. 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 [9]. In [5] the authors for the first time use the spatial network for storing the relations between aggregated views. In [1,4] materialization is characterized by workload and disk space limitation. Indices can be created on every materialized view. In order to reduce problem complexity, materialization and indexing are often applied separately. For a given space limitation the optimal indexing schema is chosen after defining the set of views to be materialized [2]. In [6] the authors proposed a set of heuristic criteria for choosing the views and indices for data warehouses. They also addressed the problem of space balancing but did not formulate any useful conclusions. [8] presents a comparative evaluation of benefits resulting from applying views materialization and data indexing in data warehouses focusing on query properties. Next, a heuristic evaluation method was proposed for a given workload and global disk space limitation.

In this paper we present a new approach to storing and processing of long aggregate lists. In our approach we materialize the calculated values (query results), but we divide the data set into smaller sets that we call pages. Our paper is organized as follows: section 2 briefly describes the motivation for our work. In section 3 all the specification and configuration aspects are presented. Section 4 describes all the most interesting details of the proposed solution, and in section 5 we present the current state of the art. In section 6 we present test results. Finally, section 7 concludes the paper.

2 Motivation

We are working in the field of spatial data warehousing. Our system (Distributed Spatial Data Warehouse – DSDW) presented in [3] 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 sent from a meter to the server contains a timestamp, a meter identifier, and the reading values. Periodically the extraction system loads the data to the database of our warehouse.

In our current research we are trying to find the weakest points of our solution. After different test series (with variations of aggregation periods, numbers of telemetric objects etc.) we found that the most crucial problem is to create and manage long aggregate lists. The aggregate list is a list of meter reading values aggregated according to appropriate time windows. A time window is the amount of time in which we want to investigate the utility consumption. The aggregator is comprised of the timestamp and aggregated values.

When we want to analyze utility consumption we have to investigate consumption history. That is when the aggregate lists are useful.

In the system presented in [3] aggregate lists are used in the indexing structure that is a modification of an aR-Tree [7]. Every index node encompasses some part of the region where the meters are located and has as many aggregate lists as types of meters featured in its region. If there are several meters of the same type, the aggregate lists of the meters are merged (aggregated) into one list of the parent node.

The aggregate lists are stored in the main computer memory. Memory overflow problems may occur when one wants to analyze long aggregation periods for many utilities meters. If we take into consideration the fact that the meter readings should be analyzed every thirty minutes, simple calculations reveal that the aggregate list grows very quickly with the extension of an aggregation period. For instance, for single energy meter an aggregate list for one year has $365 \cdot 48 = 17520$ elements. Each of the aggregators creating the list stores a few values, so the memory consumption is high. In order to prevent memory overflows we designed a memory managing algorithm applied in the system presented in [3]. The mechanism defines a memory limit when the system starts. The limit is always checked before some new aggregate list is created. If upon being loaded a new list threatens to exceed a limit, the mechanism searches for a less frequently read node in the indexing structure and removes its aggregate lists from the memory, providing space for the new lists. The mechanism performs well when system uptime is not long. The creation and removal of aggregate list produces memory fragmentation that results in memory overflow errors, even though the memory limit had not been exceeded. Hence we decided to search for a new approach to storing and processing aggregate lists with no length limitations. Our main objectives were: the solution must be efficient and scalable, applicable in indexing structures such as aR-tree and easy to use. We named the solution a Materialized Aggregate List (MAL).

3 Specification

The main idea of the proposed solution is to provide a user with a simple interface based on the standard Java list mechanism – a set of two functions: $hasNext()$ and $next()$ which permits the convenient browsing of the list contents. Our purpose was to create a list that could be used as a tool for mining data from the database as well as a component of indexing structure nodes (fig. 1). Below we present an example showing how the list can be used in the program code.

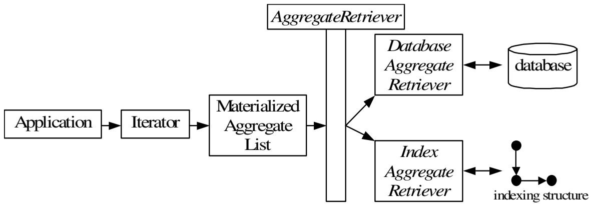
AggregateRetriever
Application
Iterator
Materialized Aggregate List
Database Aggregate Retriever
database
Index Aggregate Retriever
indexing structure

Fig. 1. MAL idea – provide a solution based on a well-known standard

```
(1) MALList list = new MALList(categ, ob, dbConn);
(2) Iterator iterator = list.iterator(startDate);
(3) while (iterator.hasNext()){
(4)   Aggregator a = (Aggregator)iterator.next();
(5)   /* use theaggregator  */
(6)   /* time condition breaking the iteration */
(7) }
(8) list.close();
```

In the first line we see how the list object is constructed. The constructor parameters are: the category (defines list type), an identifiable object (spatial telemetric object or indexing structure node) and a database connector.

The second line creates the iterator that allows list browsing. In the standard Java implementation the iterator function has no parameter. In the case of MAL there is one parameter defining the timestamp of the first aggregator returned by the $next()$ function call.

Lines 3-7 contain instructions known from the standard Java solution. First (line 3) it checks the availability of the next element in the list; the element is retrieved (line 4) and some operations using this element are performed (line

5). A time condition can be put in the next line, breaking the iteration. The condition may be applied if it is not necessary to browse the whole list.

Line 8 closes the list. No new iterators can be created, the list waits for all running threads to complete.

4 MAL Details

As mentioned before, our main intention when designing the MAL was to build a solution free of memory overflows which would allow aggregate list handling with no length limitations. We applied the following approach: every list iterator consists of a table divided into pages. When an iterator is created some of the pages are filled with aggregators (which pages and how many is defined by the applied page-filling algorithm, see description below). The next pages are filled (the aggregators are retrieved from the iterator table), while the list is being browsed. After the whole page is read, it is refilled with new data. The solution also uses an aggregates materialization mechanism that strongly speeds up the aggregates retrieval. The most crucial configuration aspects are: the number of pages, the size of a single page, the number of available database connections and the pages-filling algorithm.

The actual list operation begins when a new iterator is created (*iterator*() function call). A new table is created and two values are calculated:

– border date. The border date is used for managing the materialized data. The border date is calculated by repeatedly adding to the install date (defined in category block of the configuration file) a width of aggregation window multiplied by the size of the table page. The date is equal to the timestamp of the first aggregator in the page.
– starting index. In the case that starting date given as a parameter in the *iterator*() function call is different from the calculated border date, the iterator index is adjusted so that a the first *next*() function call returns the aggregator with the timestamp nearest to the given starting date.

4.1 Page-Filling Algorithms

As a new iterator is constructed some of its table pages are filled with aggregators. Which pages and how many of them depends on the used page-filling algorithm. All the algorithms create the page-filling threads that operate according to the following steps:

1. Check whether some other thread filling a page with an identical border date is currently running. If yes, register in the set of waiting threads.
2. Get a database connection from the connection pool.
3. Check if the required aggregates were previously calculated and materialized. If yes, restore the data and go to 5.
4. Create the aggregate list. Materialize the list.
5. Release the database connection.

6. Browse the set of waiting threads for threads with the specified border date. Transfer the data and notify them.

In the subsections below we present three different page-filling algorithms.

Algorithm SPARE. 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 in the *next*() function if the current page (let's mark it n) is exhausted. If the last aggregator from the n page was retrieved, the algorithm calls the page-filling function to fill the $n + 2$ page while the main thread retrieves the aggregates from the $n + 1$ page. One page is always kept as a "reserve", being a spare page. This algorithm brings almost no overhead – only one page is filled in advance. If the page size is set appropriately so that the page-filling and page-consuming times are similar, the usage of this algorithm should result in fluent and efficient list browsing.

Algorithm RENEW. When the RENEW algorithm is used, all the pages are filled during creation of the new iterator. Then, as the aggregates are retrieved from the page, the algorithm checks if the retrieved aggregator is the last from the current page (let's mark it n). If the condition is true, the algorithm calls the page-filling function to refill the n page while the main thread explores the $n + 1$ page. Each time a page is exhausted it is refilled (renewed) immediately. One may want to use this algorithm when the page consuming time is very short (for instance the aggregators are used only for drawing a chart) and the list browsing should be fast. On the other hand, all the pages are kept valid all the time, so there is a significant overhead; if the user wants to browse the aggregates from a short time period but the MAL is configured so that the iterators have many big pages – all the pages are filled but the user does not use all of the created aggregates.

Algorithm TRIGG. During new iterator creation by means of the TRIGG algorithm, only the first page is filled. When during n page browsing the one before last aggregator is retrieved from the page the TRIGG algorithm calls the page-filling function to fill the $n + 1$ page. No pages are filled in advance. Retrieving the next to last aggregator from the n page triggers filling the $n + 1$ page. The usage of this algorithm brings no overhead. Only the necessary pages are filled. But if the page consumption time is short the list-browsing thread may be frequently stopped because the required page is not completely filled.

4.2 Connection Pool

A very important aspect of the Materialized Aggregate List operation is database access. The page-filling threads use the database connection for creating an aggregate list and for list materialization and restoring. The connection can be used by only one thread at a time. The connection retrieving operation may cause some threads to stop when the number of concurrently running threads is greater than the number of available connections. To optimize connection management we decided to use the concept of connection pool (generally: resource pool) and connection factory (generally: resource factory). The pool parameter is the maximal number of connections that can be obtained from the pool. After creating, the pool does not contain any connections. During application operation, any thread that requires a database connection calls a pool method for retrieving a connection. Depending on the pool state the following operations are performed:

– if the pool contains a free connection, the connection is assigned to the calling thread,
– if the pool does not contain a free connection, but the connections limit is not exceeded, a new connection is created by means of the connection factory and assigned to the calling thread
– if the pool does not contain a free connection and creating a new connection would cause the connections limit to exceed, the calling thread is stopped until some connection is returned to the pool or the pool is destroyed.

When a thread completes the operations requiring database connection, the connection is returned to the resource pool. If some threads are waiting for a connection, one of them will be assigned a connection and notified.

4.3 Materialization

In the presented operation of the page-filling function, points (3) and (4) mention a concept of materialization. We introduced the materialization mechanism in the DSDW system presented in [3] and the tests revealed the mechanism extreme efficiency. The idea is to store once calculated aggregators as binary data in the database, using the BLOB table column. In the current approach we use a table with three columns storing the following values: the object identifier (telemetric object or indexing structure node), page border date and aggregators in binary form. The page materialization mechanism operates identically for each page-filling algorithm.

5 State of Art and Future Plans

After finishing work on theoretical concepts we started implementation of our solution. The current state of the art contains a full implementation of the database iterator (the iterator for retrieving aggregates from a database) and all three page-filling algorithms. The list operation is convergent with the description presented in section 3. The iterator retrieving aggregates from the database can automatically process new data added by the extraction process. If some page was materialized but it is not complete (not all necessary data was found in the database when it was being filled), then the page-filling thread starts exploring the database from the point where the data was not available. The aggregates retrieving finishes if there is no more available data, then the $hasNext()$ function call returns false.

We are nearing completion of the work on the MAL iterator, which permits us to apply the solution in indexing structures (such as aR-Tree). The applied page-filling algorithm is very similar to the TRIGG algorithm for the database iterator.

The data warehouse structure described in [3] applies distributed processing. We also suppose that in this aspect introducing the MAL to our system will bring benefits in efficiency. The current approach to sending complete aggregate lists as a partial result from a server to a client results in high, single client module load. When we divide the server response into MAL pages, the data transfer and the overall system operation will presumably be more fluent.

6 Test Results

This section contains a description of the tests performed with the current implementation of the presented solution. The tests were executed on a machine equipped with Pentium IV 2.8 GHz and 512 MB RAM. The software environment was Windows XP Professional, Java Sun 1.5 and Oracle 9i. The tests were performed for all three page-filling algorithms. Each of the algorithms was applied in the iterator used for retrieving aggregates from the database for 3, 6, 9, and 12 months. The aggregates were created with a time window of 30 minutes. The created aggregates were not used in the test program; the program only sequentially browsed the list. Aggregates browsing was performed twice: during the first run the list has no access to the materialized data, and during the second run a full set of materialized data was available. The MAL parameters, page number, page size and the number of available database connections, had the following values:

- page size: 48 (1 day), 240 (5 days), 336 (7 days), 672 (14 days), 1008 (21 days), 1488 (31 days – 1 month), 2160 (46 days – 1.5 month), 2976 (62 days – 2 months) and 4464 (93 days – 3 months),
- page number: $2 \div 10$.
- number of database connections: $1 \div pageNumber + 1$

Our goal was to find the best combination of the MAL parameters: the page-filling algorithm, number of pages, size of a single page and number of available database connections. The choice criterion consisted of two aspects: the efficiency measured as a time of completing the list-browsing task and memory complexity (amount of the memory consumed by the iterator table).

6.1 Page Size and Page Number

We first analyze the results of completing the list-browsing task during the first run (no materialized data available) focusing on the influence of the page size parameter. We investigated the influence for various numbers of pages and for all three algorithms always setting the number of available database connections to 1. We observe that for all three algorithms the influence is very similar; graphs of

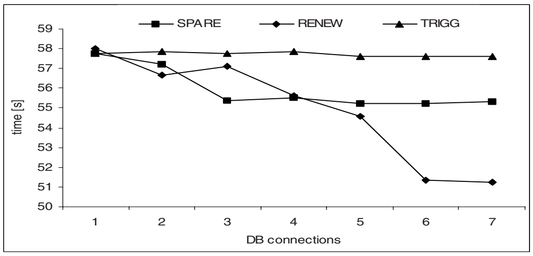

Fig. 2. Operation of the SPARE algorithm for retrieving aggregates for 6 months

the relations are very convergent. Figure 2 presents a graph showing the results obtained for the SPARE algorithm for the aggregation period of 6 months. The list browsing times for small pages are very diverse. For the presented results the times for a page of size 48 vary from 30 to 160 seconds depending on the amount of pages. MAL operation for a page of size 240 is much more stable; the differences resulting from the different number of pages do not exceed 25 seconds. In graph we observe that for pages greater or equal 672 the list browsing time does not significantly depend on the number of pages. We must notice that the page size strongly influences the amount of memory consumed by the iterator. Hence, considering the fact that further increasing the page size brings almost no time benefit, we chose the page size 672 as the most optimal.

As next, we analyzed the influence of the combination of two parameters: number of pages and number of available database connections on the MAL efficiency. We performed the test for 1 to $number\_of\_pages + 1$ available connections because in some particular cases the MAL instance also utilizes a database connection. Again, we must notice, that number of pages influences the amount of consumed memory as well as the CPU workload (in the worst case the number of pages equals the number of concurrently running threads). Analyzing test results we concluded the following: the most optimal benefit/cost ratio is when the list is configured to work with $4 \div 6$ pages and the connection pool contains as many connections as there are pages.

6.2 Page-Filling Algorithm

After choosing the optimal parameters, we compared the time efficiency of the page-filling algorithms. Figure 3 shows a graph comparing efficiency of the algorithms for browsing the list of aggregates for 12 months. The list was configured to use 6 pages, each of size 672. The obtained results are strictly coherent with the theoretical assumptions of the page-filling algorithms. When only one database connection is available there is no time difference in the operation of the algorithms. But along with increasing the number of available connections the SPARE and the RENEW algorithms show better efficiency while the TRIGG algorithm efficiency remains unchanged. The TRIGG algorithm fills only one page at a time; it uses only one database connection. As a result, increasing the

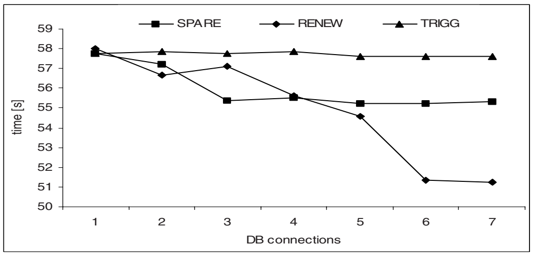

Fig. 3. Comparison of the page-filling algorithms

number of available connections brings no time profit. The STEPS algorithm launches at maximum 2 threads concurrently, utilizing at most 3 database connections what is clearly seen in the graph. And finally, the RENEW algorithm fills all the pages concurrently, utilizing all the available connections. It improves its efficiency each time the number of database connections increases. We chose this algorithm as the most efficient one.

Therefore, to summarize the parameters selection we can state that the MAL works efficiently for the following configuration: the RENEW algorithm, number of pages 4 ÷ 6, size of a single page 672, number of available database connections equals number of pages.

6.3 Materialization

The aspect last investigated was materialization influence on system efficiency. The results interpretation reveals that materialization strongly improves system efficiency. In figure 4 there is a graph showing the MAL operation for the TRIGG algorithm for various number of pages of sizes 672 and 1488 and with one database connection. As the first run we marked the list operation with no materialized data, and as a second run we marked the operation with the full set of materialized data. In both page size variants the benefit of materialization is very similar, and upon analyzing the charts, we can state that using the materi-

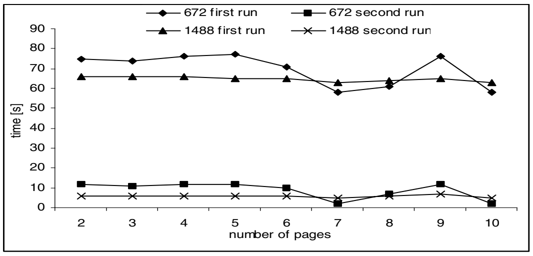

Fig. 4. Comparison the first and second run for the TRIGG algorithm with pages containing 672 and 1488 aggregators

alized data the list operates from 5 to 8 times faster than when no materialized is used. A similar situation can be observed for all the page size and page number parameter combinations.

7 Conclusions

In this paper we presented the Materialized Aggregate List (MAL). The MAL is a data structure for storing long aggregate lists. The list can be applied as a component of indexing structure nodes in indexes like an aR-Tree. The aggregators stored in the list can be retrieved from both the database and from other levels of an indexing structure. In our solution we applied the idea of aggregates materialization. The materialization has a very strong, positive influence on list efficiency. We presented the current state of the art, our future plans, and the theoretical and practical details of our solution. The paper additionally describes results of the preliminary tests.

References

1. Baralis E., Paraboschi S., Teniente E.: Materialized view selection in multidimensional database. In Proc. 23th VLDB, pages 156-165, Athens, 1997.
2. Golfarelli M., Rizzi S., Saltarelli E.: Index selection for data warehousing. In. Proc. DMDW, Toronto, 2002.
3. Gorawski M., Malczok R.: Distributed Spatial Data Warehouse Indexed with Virtual Memory Aggregation Tree. 5th STDBM_VLDB'04 Workshop, Toronto 2004
4. Gupta H.: Selection of views to materialize in a data warehouse. In. Proc. ICDT, pages 98-112, 1997.
5. Harinarayan V., Rajaraman A., Ullman J.: Implementing data cubes efficiently. In. Proc. ACM SIGMOD Conf., Montreal, 1996.
6. Labio W.J., Quass D., Adelberg B.: Physical database design for data warehouses. In. Proc. ICDE, pages 277-288, 1997.
7. Papadias D., Kalnis P., Zhang J., Tao Y.: Effcient OLAP Operations in Spatial Data Warehouses. Spinger Verlag, LNCS 2001
8. Rizzi S., Saltarelli E.: View Materialization vs. Indexing: Balancing Space Constraints in Data Warehouse Design, CAISE, Austria 2003
9. Theodoratos D., Bouzehoub M.: A general framework for the view selection problem for data warehouse design and evolution. In. Proc. DOLAP, McLean, 2000