



# Towards storing and processing of long aggregates lists in spatial data warehouses

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 comparison of two approaches for storing and processing of long aggregates lists in a spatial data warehouse. An aggregates list contains aggregates, calculated from the data stored in the database. Our comparative criteria are: the efficiency of retrieving the aggregates and the consumed memory. The first approach assumes using a modified Java list supported with materialization mechanism. In the second approach we utilize a table divided into pages. For this approach we present three different multi-thread page-filling algorithms used when the list is browsed. When filled with aggregates, the pages are materialized. We also present test results comparing the efficiency of the two approaches.

## 1 Introduction

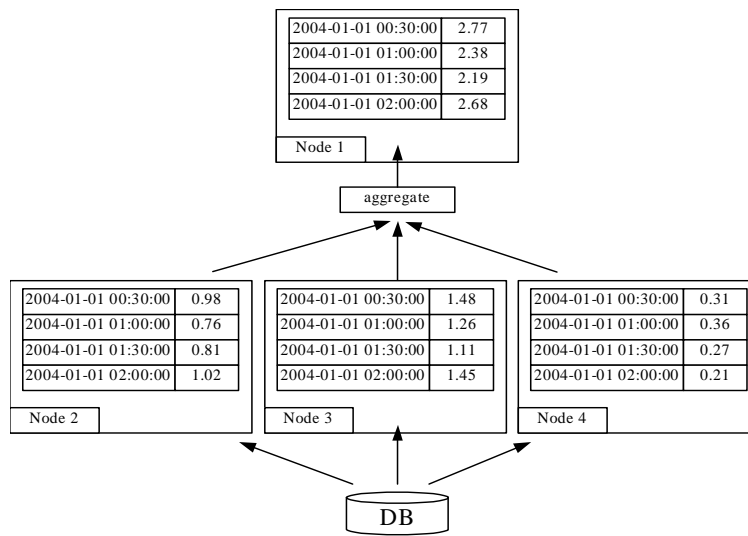
Data warehouses store and process huge amounts of data. We are working in the field of spatial data warehousing. Our system (Distributed Spatial Data Warehouse—DSDW) presented in [3] is a data warehouse gathering and processing huge amounts of telemetric information generated by the telemetric system 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. The data gathered in the database provide information about a given utility consumption. Thanks to this information we can analyze an average consumption of a given medium and appropriately control its production and distribution. When we want to analyze utility consumption we have to investigate consumption history. That is when the aggregates lists are useful. In case of electrical energy providers, meter reading, analysis, and decision making is highly time sensitive. For example, in order to take stock of energy consumption all meters should be read and the spatial data analyzed every thirty minutes. The data warehouse operation must be interactive. 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 [1, 5] materialization is characterized by workload and disk space limitation. Indexes 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 our current research we are trying to find the ways to improve the DSDW efficiency and scalability. 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 aggregates lists. The aggregates 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 (fig. 1). In the system presented in [3] aggregates lists are used in the indexing structure,

TIMESTAMP	ZONE 1	ZONE 2
2004-01-01 00:30:00	0.786	0.13
2004-01-01 01:00:00	0.97	0.65
2004-01-01 01:30:00	1.034	0.453
• • •		

**Fig. 1.** Example of an aggregates list for time window of 30 minutes.

aggregation tree, 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 aggregates lists as types of meters featured in its region. If there are several meters of the same type, the aggregates lists of the meters are merged (aggregated) into one list of the parent node (fig. 2). In the following sections we present and compare two different approaches



**Fig. 2.** A hypothetical indexing structure.

for storing and managing long aggregates lists. First we present the theoretical background and then we discuss the efficiency test results. Finally we conclude the paper choosing the more efficient and scalable solution.

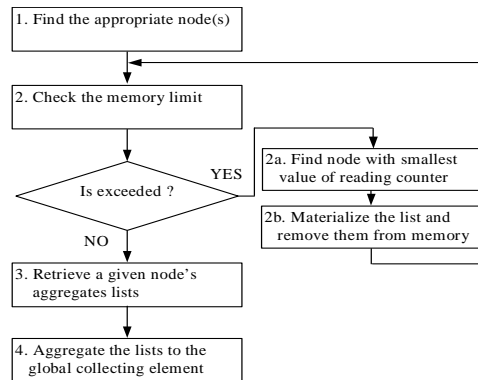
## 2 First approach—materialized Java list

The first approach assumes using a standard Java language list (like *LinkedList* or *ArrayList*, see [10]) to store the aggregates. Created aggregates are added to the list; the list is sorted according to the timestamp. The aggregates 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 aggregates list grows very quickly with the extension of an aggregation period. For instance, for single energy meter an aggregates list for one year has  $365 \cdot 48 = 17520$  elements. In order to protect the system from this failure we designed a memory management algorithm. The algorithm operation is unnoticeable for a system user.

When the system starts, the algorithm evaluates the memory capacity by creating artificially generated aggregates lists. Each allocated aggregates list is counted. The action is continued until there is no more free memory. The next step is to remove the allocated lists and compute the value indicating the maximal amount of lists in the memory during system operation. That value is computed as 50% of the all of allocated

lists. The 50% margin is set in order to leave enough memory for other system needs, e.g. user interface and insurance against memory fragmentation.

Memory overflow may occur when new aggregates lists are created. When the previously mentioned limit is exceeded (too many aggregates lists in the memory) then some aggregates lists have to be removed from the memory. In order to determine which lists to remove, we applied a mechanism that uses a lists reading counter. Every indexing structure node has the reading counter increasing each time the lists are read. The memory managing mechanism searches for the nodes with the smallest value of the reading counter and removes them from the memory. Aggregates removal is proceeded by the materialization operation. In figure 3 we present a flowchart illustrating the memory managing algorithm operation supported by the materialization mechanism.



**Fig. 3.** The memory managing algorithm operation.

## 2.1 Materialization

In order to save time spend on raw data processing we decided to apply the idea of aggregates lists materialization. When a given aggregates list is created for the first time, a set of queries must be executed and the resulting raw data processed. Once calculated, the data is stored for further use. The materialization mechanism combines the Java streams and the Oracle BLOB table column. The created aggregates are stored in a table consisting of two columns, the first being NUMBER storing node's identifier and the second is BLOB storing materialized binary data of a given node's aggregates lists. Figure 4 shows a simple schema of the materialization mechanism. The presented approach of storing a single aggregates list as one stream has two main drawbacks: the list must always be read starting from the beginning what, if not necessary, can be very time-expensive, and the list cannot be easily updated (to attach new aggregators we must restore, update and store whole long list). In the next section we present a different solution to the list materialization problem.

## 3 Second approach—Materialized Aggregate List

The second approach is called a Materialized Aggregate List (MAL). Our main goal when designing the MAL 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. 5). In this paper we focus mainly on the differences between the previously and currently applied solutions. For more theoretical and practical details on the Materialized Aggregate List please refer to [4]. The list interface is identical to the standard Java list so the MAL can easily replace the previously described aggregates list. We also suppose that the applied multi-thread page-filling algorithms and selective materialization will result in the MAL being more efficient when compared to the standard Java list. Our main intention when designing the MAL was to build an efficient solution free of memory

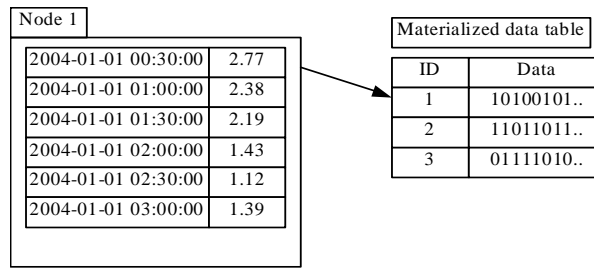


Fig. 4. A simple schema of the materialization mechanism.

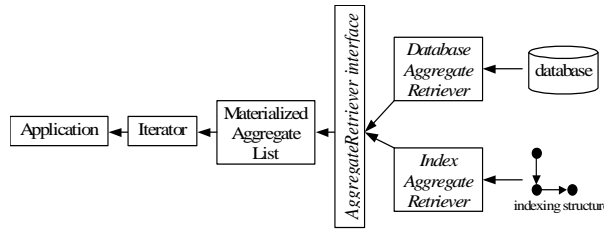


Fig. 5. MAL idea – provide a solution based on a well-known standard.

overflows which would allow aggregates list handling with no length limitations. The MAL structure and operation is based on the following approach: every list iterator uses a table divided into pages (the division is purely conventional). 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). Applying a multi-thread approach allows filling pages while the list is being browsed. The solution also uses an aggregates materialization mechanism to store once created aggregates.

The actual list operation begins when a new iterator is created (*iterator()* function call). Every iterator is characterized by two dates:

- border date. The border date is used for managing the materialized data. The border 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.

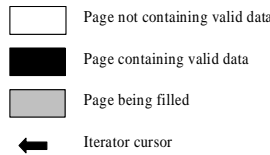
Consider the following: we have the install date 2004-01-01 00:00:00, an aggregation window width of 30 minutes and page size of 240. So, as can be easily calculated, on one page we have aggregates from five days ( $48$  aggregates from one day,  $\frac{240}{48} = 5$ ). Next we create an iterator with the starting date 2004-01-15 13:03:45. The calculated border date will be 2004-01-11 00:00:00, starting index 218 and the first returned aggregator will have the timestamp 2004-01-15 13:30:00 and will contain the medium consumption between 13:00:00 and 13:30:00.

### 3.1 Page-filling algorithms

The iterator table pages are filled by separate threads. Regardless of the applied page-filling algorithm an individual thread, characterized by a page border date, operates according to the following steps:

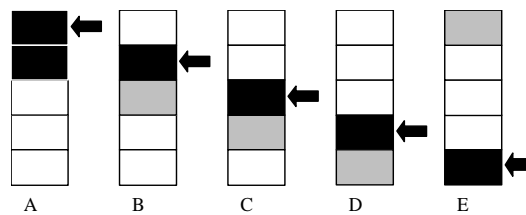
1. Check whether some other thread filling a page with an identical border date is currently running. If yes, register in the set of waiting threads and wait.
2. If no, check if the required aggregates were previously calculated and materialized. If yes, restore the data and go to 4.
3. If no, fill the page with aggregates. Materialize the page.
4. 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 used for retrieving aggregates from the database. Their operation is illustrated with figures that use the symbols described in figure 6.



**Fig. 6.** Symbols used in the page-filling algorithms descriptions.

**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 (fig. 7). 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.



**Fig. 7.** Operation of the SPARE algorithm.

**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 (fig. 8). 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 (fig. 9). 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.

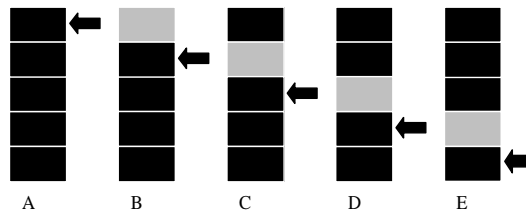


Fig. 8. Operation of the RENEW algorithm..

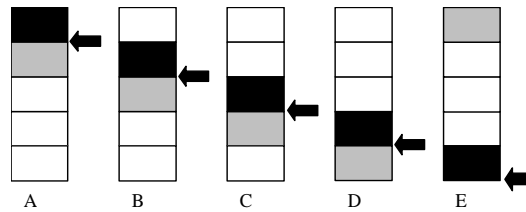


Fig. 9. Operation of the TRIGG algorithm.

### 3.2 MAL in indexing structure

The idea of the page-filling algorithm in the MAL running as a component of a higher-level node is the same as in the TRIGG algorithm for the database iterator. The difference is in aggregates creating because the aggregates are created using aggregates of lower-level nodes. The creation process can be divided into two phases. In the first phase the aggregates of the lower-level nodes are created. This operation consists of creating the aggregates and materialization. In the second phase, the aggregates of the higher-level node are created through merging all the available materialized pages of the lower-level nodes created in the first phase into pages of the higher-level node. The second phase is performed using the materialized data created in the first phase and its execution takes less than 10% of the whole time required for creating the aggregates of the higher-level node.

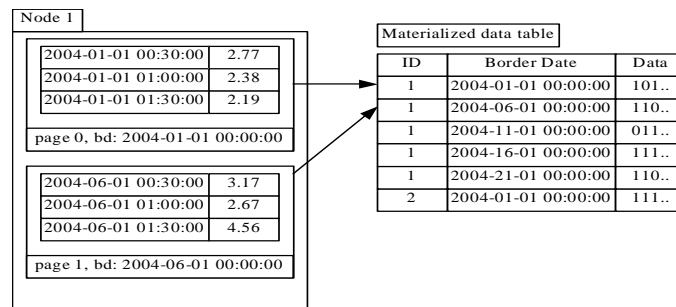
To control the number of concurrently running threads we use a resource pool storing the iterator tables. Thanks to such approach we are able to easily control the amount of memory consumed by the system (configuring the pool we decide how many tables it will contain at maximum) and the number of running threads (if no table is available in the pool a new iterator will not start its page-filling threads until some other iterator returns a table to the pool).

### 3.3 Materialization

Page-filling thread operation description shows that the MAL applies materialization. For the MAL 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 (fig. 10). The page materialization mechanism operates identically for each page-filling algorithm. The MAL can automatically process new data added by the extraction process. If some page was materialized but it is not complete, then the page-filling thread starts retrieving aggregates from the point where the data was not available.

## 4 Test results

This section contains a description of the tests performed for the both presented approaches. The aggregates were created for 3, 6, 9, and 12 months 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 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.



**Fig. 10.** A schema of the materialization mechanism applied in the MAL.

In the case of the first presented approach we have little influence on the list configuration. But in the case of MAL we can change the following configuration parameters to make the list operation most efficient:

- page size defines how many aggregates is stored on a single list page,
- page number defines the amount of pages creating the iterator table,
- page-filling algorithm defines which algorithm is to be used for filling the pages.

In order to compare the two presented approaches we first analyze operation of the MAL to find the best combination of the configuration parameters. 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).

#### 4.1 MAL configuration

The tests were performed for the three page-filling algorithms, size of a single page varied from 48 aggregates (1 day) to 4464 (93 days – 3 months) and number of pages  $2 \div 10$ . The number of tables available in the table pool was limited to 1 because increasing this number brought no benefit.

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 number and the size of a single page. We investigated the relations between these parameters for all three algorithms, and we can state that in all the cases their influence is very similar; graphs of the relations are very convergent. The list browsing times for small pages are very diverse. For a page of size 48 the times vary from 30 to 160 seconds depending on the amount of pages. MAL operation for a page of size 240 is more stable; the differences resulting from the different number of pages do not exceed 25 seconds. For pages of size 672 and more we observe very stable operation of the MAL. A page size of 672 seems to be the best choice. Extending the page size brings very small efficiency gain but results in much more memory consumption. After choosing the best pair of page size and page number parameters, we compared the time efficiency of the page-filling algorithms. Figure 11 shows a graph comparing efficiency of the algorithms applied in the child nodes for browsing the list of aggregates for 3, 6, 8 and 12 months. The lists were configured to use 6 pages, each of size 672. In the graph we observe that the SPARE and the TRIGG algorithms show similar efficiency. Along with extending the aggregation period the operation time increases; for the TRIGG algorithm the increase is purely linear. The RENEW algorithm shows worse efficiency, especially for long aggregation periods of 6 and 12 months. Filling and merging the pages not used during the list browsing results in worse performance. Therefore, to summarize the parameters selection we can state that the MAL works efficiently for the following configuration: number of pages  $4 \div 6$ , size of a single page 672 and TRIGG as the page-filling algorithm.

#### 4.2 Efficiency comparison

In this subsection we compare efficiency of the two presented approaches. The scenario concerns operation of the lists when applied in a theoretical indexing structure. The structure consisted of one parent node and  $5 \div 20$  child nodes; the query concerned parent aggregates but obviously resulted in creating the aggregates

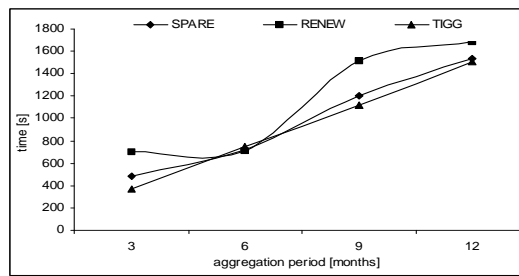


Fig. 11. Page-filling algorithms operation for index iterator

of all the child nodes. As first we compare the results of the lists operation during the first run when no materialized data was available. Figures 12 and 13 present graphs for gathering aggregates from respectively 5 and 20 child nodes. In the both cases the MAL shows better efficiency. It operates about 40% faster than the standard Java list supported by the materialization mechanism. Further graph analysis reveals that extending the aggregation period results in operation time growth. The growth is significantly greater in the case of the Java list. The conclusion is that the MAL solution is more scalable.

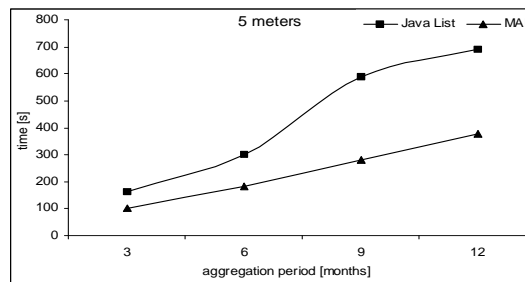


Fig. 12. Comparison of efficiency of the Java list and the MAL for 5 meters.

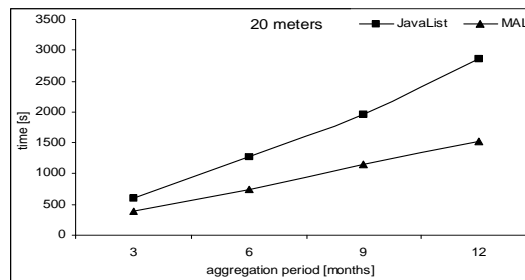


Fig. 13. Comparison of efficiency of the Java list and the MAL for 20 meters.

The aspect last investigated was materialization influence on system efficiency. The test results interpretation reveals that materialization strongly improves efficiency of both approaches. Thanks to materialization both lists operate from 6 to 8 times faster than when no materialized data is available. The longer the aggregation period and the more meter readings are to be aggregated the greater the benefit of materialization

## 5 Final conclusions and future plans

In this paper we presented a comparative study of two approaches for storing long aggregates lists in spatial data warehouse. The first approach is a standard Java list supported by aggregates materialization mechanism. The second solution is called Materialized Aggregate List (MAL) and is a data structure for storing long aggregates lists. The MAL also uses the materialization mechanism. After presenting the theoretical background for both approaches we discussed test results proving the MAL being more efficient and scalable. The MAL operates about 40% faster than a standard Java list.

The data warehouse structure described in [3] applies distributed processing. We suppose that in this aspect introducing the MAL to our system will bring benefits in efficiency. The current approach to sending complete aggregates 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. Implementation and testing of those theoretical assumptions are our future plans.

## References

1. Baralis E., Paraboschi S., Teniente E.: *Materialized view selection in multidimensional database*, In Proc. 23<sup>th</sup> 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.: *Materialized aR-Tree in Distributed Spatial Data Warehouse*, SSDA\_ECML/PKDD Workshop, Pisa 2004.
4. Gorawski, M., Malczok, R.: *On Efficient Storing and Processing of Long Aggregate Lists*, DaWaK, Copenhagen 2005.
5. Gupta H.: *Selection of views to materialize in a data warehouse*, In. Proc. ICDT, pages 98–112, 1997.
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.: *Efficient OLAP Operations in Spatial Data Warehouses*, Springer 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.
10. *Java<sup>TM</sup> 2 Platform Standard Edition 5.0, API Specification*, Sun Microsystems, 2005.