STUDIA INFORMATICA 2004
Volume 25 Number 1 (57)

Marcin GORAWSKI, Paweł MARKS
Institute of Computer Science, Silesian University of Technology

DATA LOADING BASED ON UB-TREE INDEX IMPLEMENTED IN DESIGN-RESUME/JAVABEANS ENVIRONMENT

Summary. This article describes a method of query execution time optimization based on the UB-Tree index [1] which is used for data clustering. The optimization method was implemented in a DR/JB component environment [3].

Keywords: data loading, UB-Tree, optimization, data warehouses

ŁADOWANIE DANYCH WYKORZYSTUJĄCE INDEKS UB-TREE ZAIMPLEMENTOWANE W ŚRODOWISKU DESIGN-RESUME/JAVABEANS

Streszczenie. Artykuł przedstawia metodę optymalizacji czasu wykonywania zapytań kierowanych do hurtowni danych poprzez odpowiednie rozlokowanie danych na dysku zgodnie z indeksem UB-Tree [1]. Implementacja dokonana została w środowisku komponentowym DR/JB [3].

Słowa kluczowe: ładowanie danych, UB-Tree , optymalizacja, hurtownie danych

1. Introduction

Data warehouses gather and process large amounts of data (often tens of GB). Not only proper managing of such amount of data is important but also a high level of efficiency must be held. Minimization of the queries execution time can be achieved in many different ways. These are for instance:

- indexing that speeds up searching and data joining,
- initial data aggregation, performed during the ETL process,
- materialized views,

- taking over the control of SQL optimizer,
- intentional data redundancy in dimension tables (denormalization of dimension).

Indexing allows to quickly locate tuples fulfilling the specified conditions (B*Tree index) and speed up the tables joining operation when is used for low cardinality attributes like color or sex (bitmap index) [5]. Moreover, external indexes working next to database engine are commonly used (e.g. aggregate tree). Such index is created in RAM memory so all operations using it are very fast. Initial data aggregation and materialized views technique allow to prepare required data earlier, but possible query set must be known before creating aggregates or materializing views. Using hints for SQL query optimizer can also bring a significant reduction of execution time. Denormalization of the dimension and intentional data redundancy is a very common approach in data warehouses. It allows to save a lot of time required for joining large data tables. The method of query execution time optimization presented in this paper bases on the UB-Tree index [1]. The UB-Tree clusters data in a way which can reduce number of accesses to disjoint data pages on a hard drive. This reduces both the time needed for fetching data from the disc and the query execution time.

2. Idea of the UB-Tree

UB-Tree is a multidimensional extension of the B-Tree index that is very common in many available data base systems. UB-Tree helps to manage multidimensional data. Increase of query execution efficiency is obtained by data clustering. Clustering causes that tuples with the same or similar values of attributes are stored together on the disc. This is important when dealing with point queries or range queries. Let's analyze the influence of the data distribution on the efficiency of the point query for a scalar key.

K	1	3	2	2	3	3	1	1	2	3	1	3	2	1	3	3
	Page 1				Page 2				Page 3				Page 4			

Fig. 1. Random data distribution on pages
Rys. 1. Dane losowo rozmieszczone na stronach

K	1	1	1	1	1	2	2	2	2	3	3	3	3	3	3	3
	Page 1				Page 2				Page 3				Page 4			

Fig. 2. Data sorted by a key value
Rys. 2. Dane uporządkowane wg narastającej wartości klucza

Figure 1 shows the exemplary distribution of 16 tuples with a set of key values $K \in \{1,2,3\}$ on disc pages containing 4 tuples each. Execution of the point query for a key value $K = 3$ requires loading of 4 data pages. Figure 2 presents the same data set sorted by a key attribute. In this case only 2 pages must be loaded to fetch all the tuples with $K = 3$: pages 3 and 4. Assuming that data pages don't have to be located on the disc next to each other, reading any of them involves long lasting I/O operations like head positioning and sector reading. We can see that simple data ordering may result in 50% reduction of the query execution time (reading only 2 of 4 pages).

Figures 3 and 4 show the comparison of two-dimensional data distribution. A function $(K_2, K_1) \rightarrow A$ has been defined to map values of the key attributes to so called tuple address. The function has a following definition:

- $(1,1) \rightarrow 0$
- $(1,2) \rightarrow 1$
- $(2,1) \rightarrow 2$
- $(2,2) \rightarrow 3$

The data in the figure 3 has random distribution whereas data in the figure 4 was sorted by address value A.

K_1	1	1	2	1	1	2	2	1	2	1	2	2	1	1	1	2
K_2	1	2	1	1	2	1	1	1	2	1	1	2	2	2	1	2
A	*0*	*2*	*1*	*0*	*2*	*1*	*1*	*0*	*3*	*0*	*1*	*3*	*2*	*2*	*0*	*3*
	Strona 1				Strona 2				Strona 3				Strona 4			

Fig. 3. Random distribution of two-dimensional data on pages
Rys. 3. Dane dwuwymiarowe losowe rozmieszczone na stronach

K_1	1	1	1	1	1	2	2	2	2	1	1	1	1	2	2	2
K_2	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
A	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*2*	*2*	*2*	*2*	*3*	*3*	*3*
	Strona 1				Strona 2				Strona 3				Strona 4			

Fig. 4. Two-dimensional data sorted by an address value
Rys. 4. Dane dwuwymiarowe uporządkowane wg narastającej wartości adresu

To read tuples with address $A = 0$ $(K_2 = 1, K_1 = 1)$ 4 pages must be loaded when data is not sorted, and only 2 of them when data is sorted. Reading tuples with $K_2 = 2$ and value of

K_1 causes loading of 4 pages for unsorted data, and again only 2 pages for sorted data. We can observe that simple data ordering may result in significant growth of query execution efficiency.

2.1. Mapping multidimensional space to linear space

In the examples presented in Figures 3 and 4 we used the transformation assigning a scalar value to the multidimensional attributes combination. In other words, we mapped a multidimensional space to a linear space and the transformation we use is so called mapping function. There are different transformation methods, for example: compound function, Lebesque function and Hilbert function [1]. Basing on a two-dimensional space we compare shapes of so called space filling curves [1] associated with these functions.

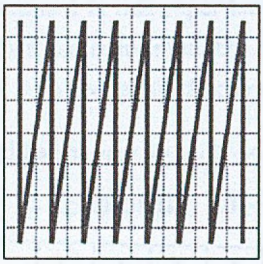

Fig. 5. Compound curve
Rys. 5. Krzywa złożona

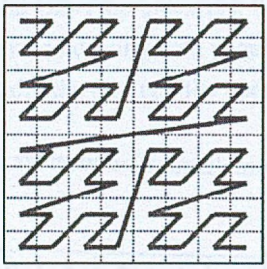

Fig. 6. Z-curve
Rys. 6. Krzywa Z

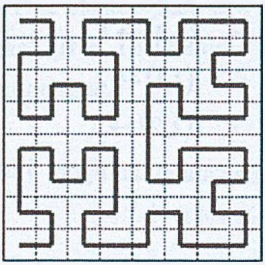

Fig. 7. Hilbert curve
Rys. 7. Krzywa Hilberta

All the curves begin in the upper left corner of the rectangle representing two-dimensional space (particular case of a multidimensional space). The upper left corner corresponds to address $A = 0$. When traversing the space along the curve, next pieces of the space are covered, and each step increments the address by 1. The first look at the shape of the compound curve (Fig. 5) lets us conclude that attributes are not treated equally and one of them has "higher priority". Such a distribution is obtained by concatenation of the binary representation of the attributes. This method is very common for keys defined using two or more attributes. The distributions of the Z-curve (Fig. 6) corresponding to Lebesque function and Hilbert curve (Fig. 7) are much better. There is none (for Hilbert curve) or a small number of significant changes of the attributes' values when the tuples' addresses increase. The main advantages of the Z-curve are symmetrical distribution in the space it fills and low complexity of the Z-address calculation algorithm.

2.2. Z-address calculation algorithm

The algorithm's input parameters are: the attributes' values vector x and the dimensions' cardinalities vector r. A few iterations are required in order to evaluate the address. During every iteration each value from the cardinality vector r is divided by 2, then it is checked whether the value is smaller than the corresponding value in the values vector x. If so, zero is stored. Otherwise, the value one is stored and the value in vector x is decreased with the value from the cardinality vector. Final Z-address is a concatenation of the stored zeros and ones. Values of the *steps()* and *steplenght()* functions are chosen in a way that causes each attribute to be divided and compared as many times as many bits contains its binary representation.

Input: $x = (x_1,...,x_d)$: d-dimensional tuple

$r = (r_1,...,r_d)$: dimensions' cardinality vector

Output: α: Z-address of a x tuple

```
for s = 1 to steps(1)
    for i = steplength(s) to 1
        if x_i < r_i/2^s then
            α_{s,d-i} = 0
        else
            α_{s,d-i} = 1
            x_i = x_i - r_i/2^s
        end if
    end for
end for
```

Fig. 8. Z-address calculation algorithm in pseudo-code notation
Rys. 8. Algorytm obliczania adresu Z w pseudokodzie

3. DR/JB component environment

A DR/JB environment [3] is a set of JavaBeans™ components, which let us create data extraction applications easily. Created applications can perform both extraction process and recovery of the interrupted extraction process basing on the Design-Resume algorithm [2]. The DR/JB environment contains the following components:

- application panel (*JDAGPanel* class),
- aggregation node (TA, *TransformationAggr* class),

- filtering node (TF, *TransformationFilter* class),
- grouping node (TG, *TransformationGroup* class),
- joining node (TJ, *TransformationJoin* class),
- sorting node (TS, *TransformationSort* class),
- extractor node (E, *FileExtractor* class),
- inserter node (I, *FileInsertor* and *DBInsertor* classes).

We can build any extraction application by connecting the above components. It will use extractors to fetch the source data, transform the data in selected transformations and finally store the transformed data in a destination place using inserter node. All components chosen by a designer create extraction graph. Its nodes are components and arrows define a data flow direction (Fig. 9).

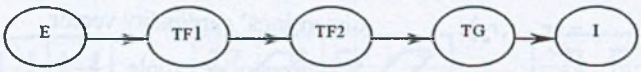

Fig. 9. Sample of the extraction graph
Rys. 9. Przykładowy graf ekstrakcji

The extraction graph is a directed acyclic graph (DAG) what means there are no loops and for each node the data flow direction is clearly defined. Each node is assigned a set of properties. They allow putting additional filters computed by the Design-Resume resumption algorithm into a graph's structure when the failure occurs. The modified graph is called a resumption graph and it is used for recovery of the interrupted extraction process.

4. The UB-loading implementation in the DR/JB environment

The main goal of the UB-loading is to load data in a way that the tuples are sorted by the calculated tuples' addresses. This results in an increase of the query processing efficiency. Implementation of the UB-Tree indexing can be done in two ways:

a) integration of the UB-Tree managing code with a data base kernel,
b) use of the specific data loading algorithm.

Integrating the UB-Tree with RDBMS is quite complicated. We need to extend definitions of the DDL statements in the SQL parser, add cost models to the query optimizer and create a library for managing the physical data distribution on the media. All these improvements result in a significant growth of the processing efficiency [1]. Controlling the loading process without any modifications to the RDBMS kernel is much easier way of increasing the efficiency. The discussed implementation bases on this approach.

The UB-loading mechanism takes advantages of the Oracle9i data base features like JDBC interface and index-organized tables (IOT). The loading process is divided into the following stages:

a) creation of the destination IOT table,
b) Z-address calculation,
c) storing tuples in the destination table,
d) creation of the indexes on the attributes used for Z-address calculation.

4.1. Creation of the destination table

The destination table is created by the following query:

```
CREATE TABLE <table_name> (
  attribute1 type1,
  attribute2 type2,
  ...
  ZINDEX NUMBER(20),
  ZINDEX_COLLISION NUMBER(10),
  PRIMARY KEY (ZINDEX, ZINDEX_COLLISION)
)
ORGANIZATION INDEX
NOLOGGING;
```

Two additional columns `ZINDEX` and `ZINDEX_COLLISION` are added to the table's definition. The `ZINDEX` column contains the tuples' Z-addresses, while the `ZINDEX_COLLISION` column was added to differ the tuples with the same Z-address. It was necessary to meet the table's unique primary key requirements. The tables are created with the `ORGANIZATION INDEX` phrase. The task of `NO LOGGING` phrase is to slightly increase efficiency of the loading by turning off logging of the table's content changes.

4.2. Z-address calculation

For each tuple the Z-address is calculated according to the calculation algorithm described in section 2.2. The input vector of the algorithm is created from the attributes chosen by an application designer. As a result of the calculation we obtain a scalar Z-address value.

4.3. Storing tuples in the destination table

Each tuple is inserted into the destination table in accordance with the algorithm presented in Figure 10. During the whole process an additional hash table `HT` is used. Before insertion it is checked if the Z-address of the currently inserted tuple is not stored in the hash table. If so, the value of the collision field (`ZIDXCOL`) assigned to the address is fetched from the table. This

is the value that has been placed in the destination table already, so before next insertion it must be increased in order to avoid repeating the value of the key K(ZIDX, ZIDXCOL). In the case the hash table does not contain the Z-address of the inserted tuple, we try to insert the tuple with collision field set to 1. In case the insertion fails, what means the collision occurred, we get the maximal value of the collision field from the destination table. After incrementing it and storing in the hash table we try to insert the tuple once again.

```
HashMap HT;
if (HT(ZIDX) != null)
    ZIDXCOL = HT(ZIDX);
else {
    if (insert(K(ZIDX,1)) == success)
        return OK;
    ZIDXCOL = getSQL("select max(ZINDEX_COLLISION) from <table>
                      where ZINDEX=" + ZIDX);
}
ZIDXCOL++;
HT(ZIDX) = ZIDXCOL;
if (insert(K(ZIDX,ZIDXCOL)) == success)
    return OK;
else
    return ERROR;
```

Fig. 10. Tuple insertion algorithm
Rys. 10. Algorytm wstawiania krotki

The hash table solves the problem of repeating of the Z-address values and reduces the number of the insertion collisions as much as possible. Presented approach holds the size of the hash table on the rational level because it stores only these addresses that caused a collision.

4.4. Creation of the indexes on the attributes used for Z-address calculation

Standard B-Tree indexes are created after loading on the attributes used for Z-address calculation. This action is not obligatory but it is possible that these attributes will be used in queries conditions since they were a part of the UB-Tree index, hence it is reasonable to speed up the access to them.

4.5. Transformation functions

Z-address can be calculated only from numeric attributes. To extend the possibility of the UB-loading usage, two transformations were defined. They allow to transform any type of attribute to the numeric value.

a) DATE transformation – changes date into a number of days since 1.January.1970,

b) ENUMERATE transformation – each new value of the attribute is assigned next integer value, colors for example: *blue* = 1, *orange* = 2, *yellow* = 3, etc.

Described mechanisms and solutions were gathered in a *DBInsertor* loading component.

5. Efficiency test of the UB-loading

We performed a few tests to examine the influence of the UB-loading on the efficiency of the whole extraction process. Researches were performed for extraction graphs of a different structure and number of processed tuples.

Test 1 and 2 were performed according to the extraction graph presented in Figure 11. The difference between them is the selectivity of the filtering transformations, which for the test 1 was 90%, and for test 2 only 40%. In both cases an extractor node read a set of 100k tuples.

Figure 12 shows the extraction graph used in the third test. Extractor fetches 150k tuples and the grouping transformation generates 1200 tuples as a result.

Test 4 is an example of processing data fetched from 2 different sources E1 and E2. Extractor E1 reads 2500 tuples and extractor E2 reads 200k tuples. The size of the result set is 2500 tuples. The extraction graph was shown in Figure 13.

Tests 5 and 6 base on the extraction graph shown in Figure 14. The difference is the number of tuples read by extractor E2, which was 100k for test 5 and 150k tuples for test 6. Sizes of the data sets read by the other sources: E1 – 2500, E3 – 1200, E4 – 500, E5 – 10. Each result set contains 2500 tuples.

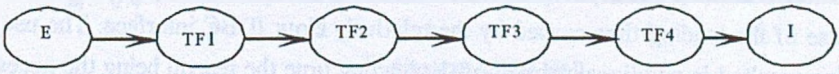

Fig. 11. Extraction graph of the tests 1 and 2
Rys. 11. Graf ekstrakcji testów nr 1 i 2

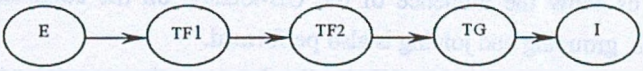

Fig. 12. Extraction graph of the test 3
Rys. 12. Graf ekstrakcji testu nr 3

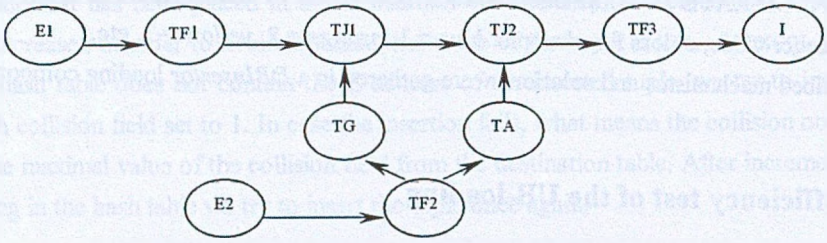

Fig. 13. Extraction graph of the test 4
Rys. 13. Graf ekstrakcji testu nr 4

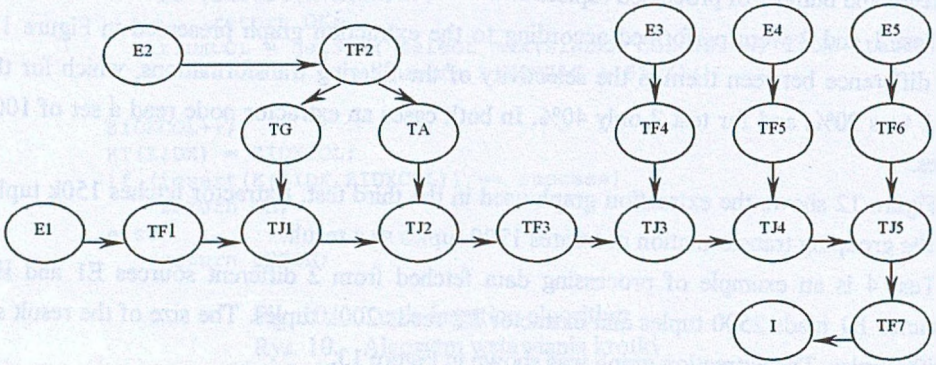

Fig. 14. Extraction graph of the tests 5 and 6
Rys. 14. Graf ekstrakcji testów nr 5 i 6

In Figure 15 we can observe a significant influence of the loaded tuples number on the total processing time in the tests 1 and 2. The result set in the test 2 (60k tuples) is approximately 6 times larger in comparison with the test 1 (10k tuples). This is connected with the increase of the loading time caused by the relatively slow JDBC interface. The use of the UB-loading resulted in additional growth of the loading time the reason being the necessity of the Z-addresses calculation and data distribution managing on the Oracle data base pages (very long-lasting operation). In the rest of the tests the result sets were smaller (1200-2500 tuples). This let us show the influence of the UB-loading on the complicated processing, where aggregation, grouping and joining is also performed.

Figure 16 shows how the use of the UB-loading increases the processing time in particular tests. In the tests 1 and 2, where the loading process is a dominant since the filtration performed in the filtering nodes has low complexity, the processing time was significantly increased. In the rest of the tests, the processing time is mostly influenced by the transformation nodes, so there is no significant change in the processing time after use of the UB-loading. In the tests performing really complex processing, the growth of the processing time does not exceed a few percent.

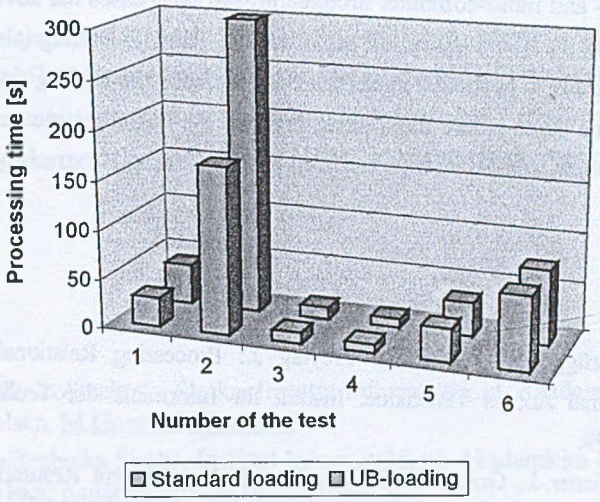

Fig. 15. Comparison of the processing time of different loading methods
Rys. 15. Porównanie czasów przetwarzania różnych metod ładowania danych

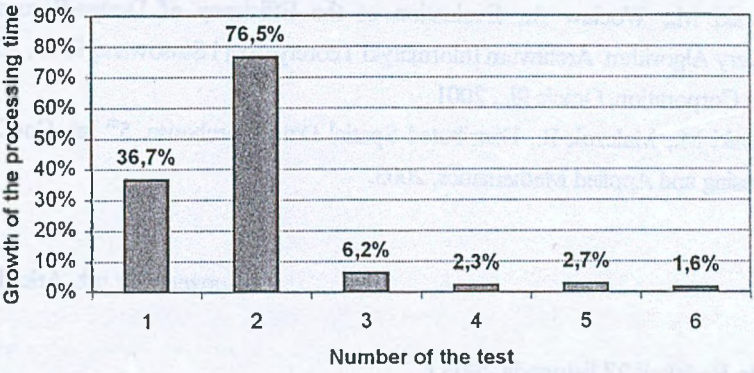

Fig. 16. Growth of the processing time during UB-loading
Rys. 16. Wzrost czasu przetwarzania po zastosowaniu ładowania UB

6. Conclusions

In this paper we present a query execution time optimizing method based on the UB-Tree index data clustering on disc pages. As described in [6] such clustering may result in increasing the query processing efficiency even three times. Clustering reduces the number of accesses to the disc what directly decreases the query processing time. The tests described in [6] were

performed in single- and multi-computer architecture. In both cases the advantage of the UB-loading was doubling the query execution performance. The UB-loading takes more time than the standard loading and in particular cases can even increase the loading time by 75% (tests 1 and 2). However, the cases where the loading process is a dominant are uncommon. During complex ETL processes, additional time required by UB-loading is extremely short.

REFERENCES

1. Markl V., Brügge B., Bayer R., Freytag J.: Processing Relational Queries using a Multidimensional Access Technique, Institut für Informatik der Technischen Universität München, 1999.
2. Labio W., Wiener J., Garcia-Molina H., Gorelik V.: Efficient Resumption of Interrupted Warehouse Loads, Stanford University, Sagent Technologies, 1999.
3. Gorawski M., Wocław A., Implementation of the Design-Resume resumption algorithm in JavaBeans technology,. Studia Informatica, No 1(52), ZN Pol. Śl., 2003.
4. Gorawski M., Wocław A., Evaluation of the Efficiency of Design-Resume/JavaBeans Recovery Algorithm. Archiwum Informatyki Teoretycznej i Stosowanej PAN, 2003.
5. Oracle Corporation, Oracle 9i, , 2001
6. Gorawski M., Malczok R., Distributed Spatial Data Warehouse. 5th Int. Conf. On Parallel Processing and Applied Mathematics, 2003.

Recenzent: Dr inż. Arkadiusz Sochan

Wpłynęło do Redakcji 27 listopada 2003 r.

Omówienie

Artykuł przedstawia metodę optymalizacji czasu wykonywania zapytań poprzez rozmieszczenie danych na dysku zgodnie z porządkiem indeksu UB-Tree. Takie działanie pozwala zredukować liczbę dostępów do stron dyskowych, a tym samym skrócić czas wykonywania zapytania. Adresy wstawianych zgodnie z algorytmem z rys. 10 krotek obliczane są wg algorytmu z rys. 8 opisanego szczegółowo w [1]. Przeprowadzone zostały testy mające na celu pokazanie, jak użycie ładowania UB wpływa na czas trwania procesu ETL. W zależności od złożoności procesu ekstrakcji zaobserwowano większe (testy 1 i 2) lub

mniejsze (testy 4, 5 i 6) wydłużenie czasu przetwarzania. Jednak wzrost czasu ładowania hurtowni danych daje zysk w postaci znacznie szybszego przetwarzania zapytań. Testy przeprowadzone w [6] wykazały, że ładowanie UB powoduje około dwukrotne zwiększenie wydajności przetwarzania zarówno w architekturze jedno-, jak i wielokomputerowej. Testy wykazały również niską wydajność interfejsu JDBC, który stanowił „wąskie gardło" systemu testowego.

Adresses

Marcin GORAWSKI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-101 Gliwice, Polska, M.Gorawski@polsl.pl

Paweł MARKS: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-101 Gliwice, Polska, p.marks@zti.iinf.polsl.gliwice.pl