Annales UMCS Informatica AI 4 (2006) 135-147

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Grouping and joining transformations in the data extraction process

Marcin Gorawski*, Paweł Marks

Institute of Computer Science, Silesian University of Technology,
Akademicka 16, 44-100 Gliwice, Poland

Abstract

In this paper we present a method of describing ETL processes (Extraction, Transformation and Loading) using graphs. We focus on implementation aspects such as division of a whole process into threads, communication and data exchange between threads, deadlock prevention. Methods of processing of large data sets using insufficient memory resources are also presented upon examples of joining and grouping nodes. Our solution is compared to the efficiency of the OS-level virtual memory in a few tests. Their results are presented and discussed.

1. Introduction

Nowadays data warehouses gather tens of gigabytes of data. The data, before loading to the warehouse, is often read from many various sources. These sources can differ in terms of a data format, so there is necessity of applying proper data transformations making the data uniformly formatted. In consecutive steps the data set is filtered, grouped, joined, aggregated and finally loaded to a destination. The destination can be one or more warehouse tables. A whole process of reading, transforming and data loading is called data extraction process (ETL).

The transformations used in the ETL process can differ in terms of complexity. A few of them are simple (e.g. filtration, projection), whereas others are very long lasting and require a lot of operational memory (e.g. grouping, joining). However, the common feature of the transformations is that each one contains at least one input and an output. This allows to describe the extraction process using a graph, whose nodes correspond to objects performing some operations on tuples, and its edges define data flow paths.

Most of commercial tools like Oracle WB do not consider internal structure of transformations and graph architecture of ETL processes. Exceptions are the research [1,2], where the authors describe ETL ARKTOS (ARKTOS II) tool. It

*Corresponding author: *e-mail address*: Marcin.Gorawski@polsl.pl

can (graphically) model and execute practical ETL scenarios, providing us with primitive expressions that brings the control over the typical tasks using declarative language. Work [3] presents advanced research on the prototypes containing the AJAX data cleaning tool.

To optimize the ETL process, there is often designed a dedicated extraction application, adjusted to requirements of a particular data warehouse system. Based on the authors' experiences [4,5], a decision was made to build a developmental ETL environment using JavaBeans components. Similar approach was proposed, in the meantime in work [6]. J2EE architecture with the ETL and ETLLet container was presented there, providing efficient ways of execution, controlling and monitoring of ETL process tasks for the continuous data propagation case.

Further speeding up of the ETL process forced us to give the JavaBeans platform up. An ETL-DR environment [7] is a successor to the ETL/JB and DR/JB [8]. It is a set of Java object classes, used by a designer to build extraction applications. These are analogous to JavaBeans components in the DR/JB environment. However, object properties are saved in an external configuration file, which is read by an environment manager object. It relieves us from recompilation of the application each time the extraction parameters change. In comparison to ETL/JB and DR/JB we improved significantly the processing efficiency and complexity of the most important transformations: grouping and joining. The possibility of storing data on a disk was added when the size of the data set requires much more memory than it is available.

In the following sections we present in detail a method of describing ETL processes using graphs and we show how this description influences the implementation. The problems resulting from the graph usage are also discussed and the methods of data processing using insufficient memory resources are presented.

2. Extraction graph

Operations performed during the extraction process can be divided into three groups:

- reading source data,
- data transformations,
- writing data to a destination.

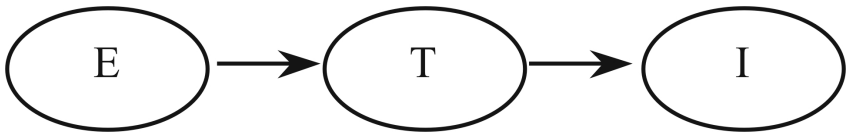

Fig. 1. One of the simplest extraction graphs. Node E is an extractor, node *T* is a transformation and node *I* is an inserter

Nodes belonging to the above mentioned operation groups are respectively: extractors (E), transformations (T) and inserters (I). From the graph point of view extractors have only outputs, transformations have both inputs and outputs, whereas inserters contain inputs only. By connecting inputs to outputs we create a connection net that defines data flow paths (Fig. 1). The data flow inside the node is possible in one direction only: from the inputs to the outputs, in the opposite direction it is forbidden. It is also assumed that connection net does not contain closed loops, which means there is no possibility to enter the same graph node traversing along the selected path of the graph. Such a net of nodes and connections is called the directed acyclic graph DAG.

3. ETL-DR data extraction environment

ETL-DR is our research environment designed in Java. It uses the extraction graph idea presented above to describe the extraction processes. During processing each graph node is associated with a thread, that is an instance of a transformation, or an extractor, or an inserter.

Available components are:

1. Extractors
   - FileExtractor (FE) – reads tuples from a source file,
   - DBExtractor (DE) – reads tuples from a database,
2. Transformations
   - AggregationTransformation (AgT) – aggregates a specified attribute,
   - FilterTransformation (FiT) – filters the stream of tuples,
   - FunctionTransformation (FuT) – user-definable tuple transformation,
   - GeneratorTransformation (GeT) – generates ID for each tuple,
   - GroupTransformation (GrT) – grouping,
   - JoinTransformation (JoT) – joining,
   - MergeTransformation (MeT) – merges two streams of tuples,
   - ProjectionTransformation (PrT) – projection,
   - UnionTransformation (UnT) – union,
3. Inserters
   - FileInserter (FI) – writes tuples to a destination file,
   - DBInserter (DI) – writes tuples to a database table via JDBC interface,
   - OracleDBInserter (ODI) – writes tuples to a database using Oracle specific SQL*Loader,
4. Specials
   - VMQueue (VMQ) – FIFO queue which stores data on a disk.

Most of the components process data on-the-fly, which means each tuple just received is transformed or analyzed independently and there is no need to gather a whole data set. The exceptions are: joining node JoT, grouping node GrT and VMQ queue.

3.1. Implementation of graph nodes interconnections

In order to facilitate analysis of interconnections between the graph nodes we have to describe the structure of inputs and outputs of the ETL-DR extraction graph nodes. Each node has a unique ID. Each node input contains ID of a source node assigned by the graph designer, and an automatically assigned number of an output channel of the source node. A node output is a multi-channel FIFO buffer with the number of channels equal to the number of inputs connected to the node (Fig. 2). When a node produces output tuples, it puts them into its output, where they are grouped into tuple packets. Upper limit of the packet size is defined by the designer. Packets are gathered in queues, separately for each output channel. The queue size is also limited to avoid unnecessary memory consumption.

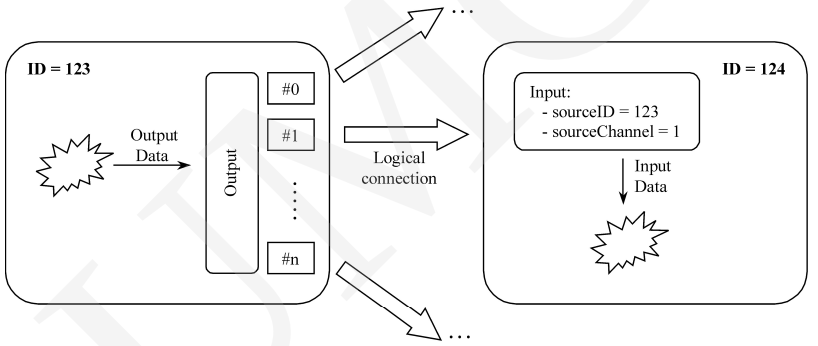

Fig. 2. Nodes interconnection on the implementation level. Data produced by the node 123 are stored in a multichannel output buffer. Source of the node 124 is defined as a node with ID = 123 and logical channel number = 1

3.2. Data exchange between nodes and a risk of deadlock

Let us analyze a case of processing performed by a part of the graph presented in Fig. 3a. The function node FuT(11) produces tuples with attributes (eID, date, transactionsPerDay), and the grouping node GrT(12) computes an average number of transactions for each employee. This is similar to the SQL query below:

```
SELECT eID, AVG(transactionsPerDay) AS avgTPD
FROM GrTFuT
GROUP BY eID
```

The joining node JoT(13) performs an action defined by the following SQL query:

```
SELECT s1.eID, s1.date, s1.transactionsPerDay, s2.avgTPD
FROM JoTFuT s1, JoTGrT s2
WHERE s1.eID = s2.eID
```

Such simple operations like grouping and joining are dangerous because they can be a reason of deadlock. This is a result of the data transferring method between node threads.

The joining node works as follows: it receives tuples from the slave input and puts them into a temporary buffer, next it receives tuples from the primary input. Each tuple from the primary input is checked if it can be joined with tuples in a temporary buffer according to the specified join condition. In the presented example, slave input is the one connected to the grouping node, as it is greatly possible, that after grouping the size of the data set will decrease and a smaller number of tuples will be kept in memory. Tuples generated by the function node are simultaneously gathered in both output channels of the node for the nodes JoT(13) and GrT(12). The grouping node aggregates data all the time, but the joining node waits for the grouped data first, and it still does not read anything from the function node. After exceeding the limit of the output queue size, the function node is halted until the queue size decreases below the specified level. This way a deadlock occurs:

- the node FuT(11) waits until the node JoT(13) starts reading data from it,
- the node GrT(12) waits for the data from the node FuT(11),
- the node JoT(13) waits for the data from the node GrT(12).

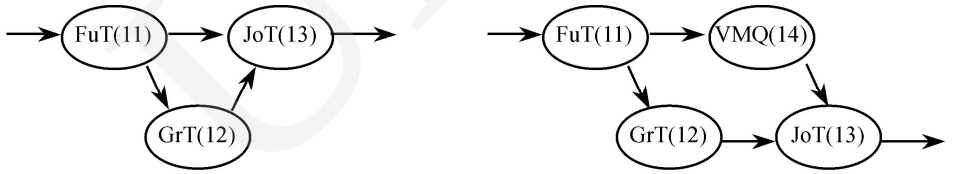

Fig. 3. Typical deadlock prone graph nodes connections (a) and a way of deadlock avoidance by the use of VMQueue component (b)

To eliminate the reason for the deadlock we have to make sure that the data from the function node FuT(11) are fetched continuously without exceeding the queue size limit. To do it we created a special VMQueue component. This is a FIFO queue with ability of storing data on a disk. It reads tuples from its input, no matter if they can be hand further or not. If tuples are fetched from the VMQ node continuously it does nothing more but transfers data from the input to the output. In the other case, it writes tuples to the disk in order to avoid overfilling of the output queue of its source node. Next, when VMQueue destination continues processing, the tuples are read from the disk and sent to the queue output. Inserting a VMQueue node between FuT(11) and JoT(13) avoids the deadlock (Fig. 3b).

3.3. Formal definition of the deadlock prone graph nodes subset

A deadlock may occur if two or more data flow paths that split in one node of the graph, meet again in another node. In other words, a given node X is connected with any of its direct or indirect source nodes by two or more paths. This let us conclude that node X must have more than one input.

Let us represent a set of source nodes of the node X as *SourceNodes(X)*, and a set of source nodes of the i-th input of X as *InputSourceNodes(X,i)*. We can define:

– $InputSourceNodes(X,i) = SourceNodes(X.in[i].sourceID) \cup \{X.in[i].sourceID\}$

– $SourceNodes(X) = \phi$ if X is an extractor,

$SourceNodes(X) = \bigcup_{i=1}^{n} InputSourceNodes(X,i)$ if X is a transformation or an inserter

– $CommonNodes(X,i,j) = InputSourceNodes(X,i) \cap InputSourceNodes(X,j)$

– $LastNode(N) = \{X \in N: SourceNodes(X) = N \setminus \{X\}\}$

If for each node X of an extraction graph, which is not an extractor, the following condition is satisfied:

$$\bigvee_{i\in[1,n]} \bigvee_{j\in[1,n]} i \neq j \Rightarrow CommonNodes(X,i,j) = \phi$$

then deadlock cannot occur. Otherwise deadlock is possible and we should use VMQueue component and insert it into the graph, to avoid the application hang. Insertion of VMQueue node makes sense only behind the nodes from a *LastNode(CommonNodes(X,I,j)*) set, that is a set of the last nodes from the set of common parts of the two data flow paths. In the example presented in the previous section it was the FuT(11) node (Fig. 3b).

3.4. Temporary data buffering on disk

During an extraction process a large number of tuples is processed. When they need to be buffered, there is a problem of selection of the right place for the buffer. Keeping them in memory is impossible because the size of the data set is usually much bigger than that of the available RAM. The only solution is storing the data on a disk. Two approaches are possible: virtual memory supported by the operating system or storing implemented on the application level in algorithms used in transformation nodes. In our ETL-DR environment the nodes using application-level virtual memory are: VMQueue, GroupTransformation and JoinTransformation.

VMQueue Component. As it was presented in Sect. 3.2 VMQueue component is a FIFO queue able to store the buffered data on a disk. Its task is to ensure the data is read from its source as it comes, even if the node receiving data from VMQueue does not work. In such a case tuples are stored in a disk file rather than put into the output buffer. Next when possible, tuples are read from the file

and hand further. Because of a sequential access to the disk file, this solution is more efficient than the OS-level virtual memory.

GroupTransformation Component. A grouping component can work in one of three modes:

1. input tuples are sorted according to the grouping attribute values,
2. tuples are not sorted, grouping in memory,
3. tuples are not sorted, external grouping.

```
procedure Group()
Begin
   List fileList;
   While Input.hasTuples() do
      Tuple T = Input.getTuple();
      If not HM.contains(Attributes(T)) then
         HM.put(Attributes(T), Aggregates(T));
      End if
      Aggregates AG = HM.get(Attributes(T));
      AG.doAggregate(T);
      If HM.size() > SIZELIMIT then
         fileList.add(WriteToFile(HM));
         HM.clear();
      End if
   End while
   AggrSource as = getSource(fileList, HM);
   Aggregates AG = null;
   While as.hasNext() do
      If AG == null then
         AG = a.next();
      Else
         Aggregates newAG = as.next();
         If (newAG.attr == AG.attr) then
            AG.aggregate(newAG);
         Else
            ProduceOutputTuple(AG);
            AG = newAG;
         End if
      End if
   End while
   ProduceOutputTuple(AG);
End
```

Fig. 4. External grouping algorithm

In case 1) aggregates are computed as they come, and memory usage level is very low. In case 2) each new combination of the grouping attributes is saved in a hash table with associated aggregates. If such a combination appears again during processing, it is located and the aggregates are updated. The number of entries in the hash table at the end of the processing equals to the number of tuples produced. Both cases 1) and 2) use only RAM.

Case 3) has features of the processing used in cases 1) and 2). First, data set is gathered in the hash table and aggregates are computed (Fig. 4). When the number of entries in the table exceeds the specified limit, the content of the table is written to the external file in the sorted order according to the grouping attribute values. Next, the hash table is cleared and the processing is continued. Such a cycle repeats until the input tuple stream ends. Then the data integration process is run. Tuples are read from the previously created files and final aggregates values are computed. This is very similar to case 1) processing with the exception of getting data from external files instead of the node input.

JoinTransformation Component. A joining node works based on the algorithm presented in Fig. 5. The first step is collecting tuples from the slave input. They can be loaded to a temporary associating array or written to a temporary disk file. Before writing to the file, tuples are sorted according to the joining attributes using the external version of the standard Merge-Sort algorithm: tuples are gathered in memory, if the limit of tuples in memory is exceeded they are sorted and written to a file. Next portions of the data set are treated in the same way. Finally, tuples from all the generated sorted files are integrated into one big sorted file. Sorting lets us locate any tuple in the external file in *log(n)* time using the binary search algorithm.

```
procedure Join()
Begin
   While Input(2).hasTuples() do
      Tuple T = Input(2).getTuple();
      HM.put(Attributes(T), T);
   End while
   While Input(1).hasTuples() do
      Tuple T = Input(1).getTuple();
      Tuple[] TT = HM.get(Attributes(T));
      For each JT in TT do
         Tuple O = Join(T, JT);
         ProduceOutputTuple(O);
      End for
   End while
End
```

Fig. 5. General joining algorithm

Additional indexing structure located in memory also decreases searching time, by reducing the number of accesses to the file. The index holds locations of the accessed tuples, which enables narrowing down the searching range when accessing consecutive tuples.

The second phase is the same, no matter if the temporary buffer is located in memory or on a disk. Only the implementation of the HM (HashMap) object changes in the algorithm presented in Fig. 5. Each tuple from the primary input is checked if it can be joined with tuples in the temporary buffer according to the specified join condition.

4. External processing tests

For tests we used data files that forced Java Virtual Machine to use much more memory than it was physically available. Tests were performed using the computer with AMD Athlon 2000 processor working under WindowsXP Professional. During tests we were changing the size of the available RAM.

4.1. Grouping test

Grouping was tested based on the extraction graph containing an extractor *FE*, a grouping node *GrT* and an inserter *I* (Fig. 6). The extractor reads the tuple stream with attributes (*eID,date,value*), in which for each employee *eID* and for each day of his work, the transaction values were saved. The number of employee transactions per day varied from 1 to 20. The processing can be described by SQL query as:

```
SELECT eID, date, sum(value) as sumVal, count(*) as trCount
FROM GrT_FE
GROUP BY eID, date
```

Fig. 6. Grouping test extraction graph

The processing time was measured depending on the number of input tuples (10, 15, 20 and 25 millions) and the type of processing. The result chart contains the total processing time (TT) and the moment of loading the first tuple to a destination, so called *Critical Time* (CT). During all the tests using external grouping (Ext) JVM was assigned only 100MB of RAM. During grouping in memory, we examined the two cases: JVM memory was set with some margin (Normal) and with a minimal possible amount of RAM (Hard) that guaranteed successful completion of the task. The obtained results are shown in Fig. 7.

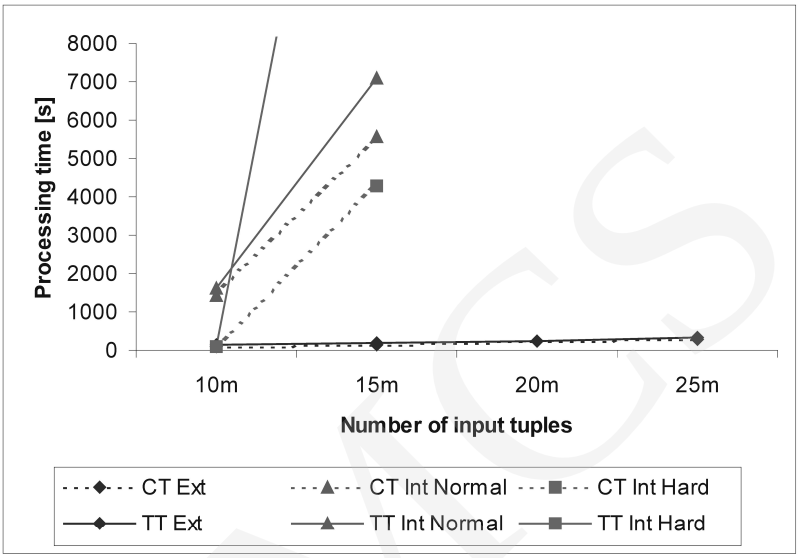

Fig. 7. Processing times measured during grouping test. *TT* is a total processing time, whereas *CT* denotes a moment when the first output tuple is produced (*Critical Time*)

The test computer contained 384MB physical RAM, and for JVM using virtual memory and for 10m and 15m tuples it was assigned respectively 450MB and 550MB during Normal test, then 300MB and 425MB during Hard test.

As it can be seen, the most efficient processing method is definitely the one using application-level data storing. Its processing time changes from 129 sec. to 322 sec. depending on the number of input tuples. The use of OS-level virtual memory causes that the whole process takes much more time. Only for 10 million of input tuples and strongly limited JVM memory, which resulted in a very low usage of the virtual memory, we obtained results slightly better than for built-in data storing. However, for 15 million tuples the processing takes an extremely long time (the line going rapidly outside the chart). The main reason for so low efficiency of a virtual memory are random accesses to the memory caused by updating aggregates in temporary buffers and Java garbage collector. The application-level storing accesses data files sequentially, and as a results this method is much more efficient.

We have not finished the OS-level virtual memory tests for 20m and 25m tuples because it needed extremely long time (several hours). Our goal was only to show that the application-level buffering can be much better than the OS-level buffering.

4.2. Joining test

Joining test is based on the extraction graph shown in Fig. 8. The extractors read the same number of tuples: *FE*1 reads tuples with attributes

(*eID,date,depID*), describing where each employee was working each day, whereas *FE*2 reads a set of tuples produced in the previous test (*eID,date,sumVal,trCount*). Joining attributes are (*eID,date*) and processing times were measured for the following number of input tuples from each extractor: 10, 15 and 20 millions.

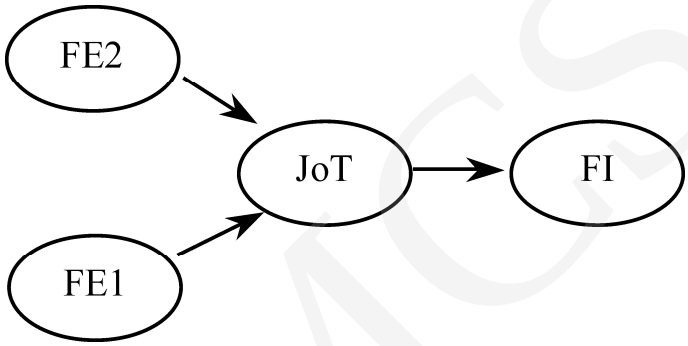

Fig. 8. Joining test extraction graph

During the test the computer was equipped with 256MB RAM, JVM was assigned 100MB RAM when joining with data storing on disk was used, and respectively 400MB and 600MB for 10 and 15 million of tuples when using virtual memory. In this test we can still observe benefits of using application-level data storing, but the difference in comparison to OS virtual memory is not so big as in the grouping test because this time the external file is accessed randomly, not sequentially. The obtained results are presented in Fig. 9.

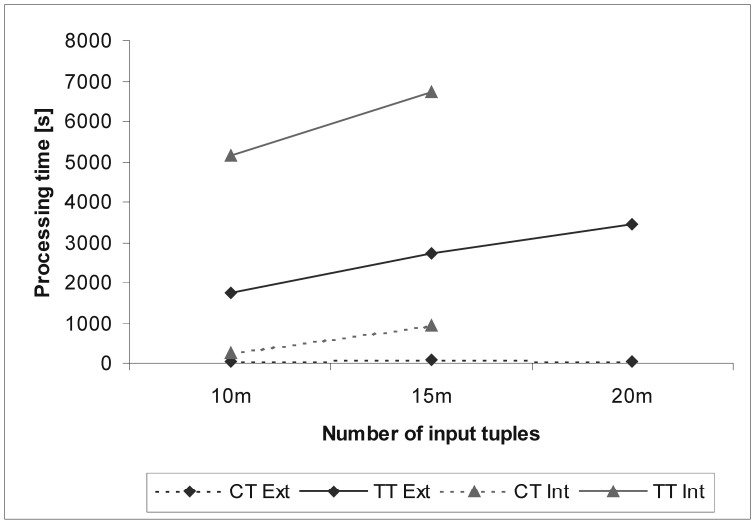

Fig. 9. Processing times measured during joining test. *TT* is a total processing time, whereas *CT* denotes a moment when the first output tuple is produced (*Critical Time*)

4.3. Real extraction test

We also performed a real extraction test. The ETL process generates a star schema data warehouse containing a fact table and two dimensions. In this test, both grouping and joining nodes appear in the extraction graph and they run concurrently: when the grouping node GrT(2) produces output tuples, the joining node JoT(30) puts them into its internal buffer (memory or a disk file). This test lets us examine the behavior of the buffering techniques when more than one node require a lot of memory resources.

The size of the input data set was 300MB. JVM required 475MB RAM to complete the task using virtual memory, and only 100MB when using application-level data storing. The computer had 256MB RAM. The ETL process using data storing took only 26 minutes, whereas when using the virtual memory, it needed 3 hours to complete only 10% of the whole task (the whole processing could take even 30 hours). Continuation of the test did not make sense, because we could already conclude that in this case the efficiency of the virtual memory was extremely low.

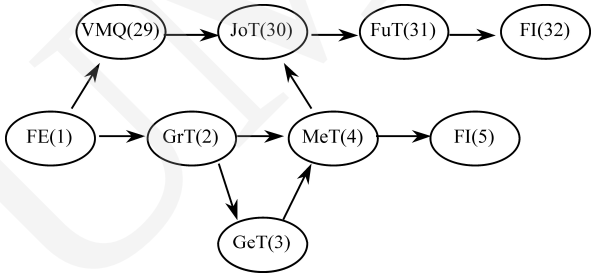

Fig. 10. The main part of the extraction graph generating star schema data warehouse. Path FE(1)-FI(32) generates fact table, whereas path FE(1)-FI(5) is responsible for producing one of the dimension tables. Extractor FE(1) reads 300MB data file

In our opinion the obtained results come from of the random accesses to the VM swap file. When many nodes keep a lot of data in a virtual memory and access it randomly (because each node runs as an independent thread) the swap file has to be read and written very often from various locations. This does not take place during application-level buffering, the external files are accessed sequentially if only it is possible (depending on the algorithm that is used).

5. Conclusions

This paper presents a concept of describing extraction processes using graphs, the meaning of graph nodes and the graph edges in the extraction process. We focused on a few implementation aspects like interconnections between nodes and the possibility of deadlock occurrence when particular graph structures are used. A method of avoiding deadlocks was also presented and it was described

by mathematical formulas. Next we introduced algorithms for external data queuing, groping and joining.

Although not tested in this paper, the presented data queuing is the efficient method of avoiding deadlocks that may occur in our ETL-DR extraction environment due to the data transferring method we used. The grouping transformation can process data sets of any size, the only limitation is the available temporary disk space. It makes use of the additional tuple stream properties, such as sorted order according to the values of the grouping attributes. The joining transformation can also process an unlimited number of tuples. It can store its slave-input tuples to disk files in a sorted order and then access any tuple in the file in log(n) time.

Our research proves that a virtual memory offered by operating systems is not always the efficient solution. Dedicated algorithms of storing data in external files working on the application level are more efficient due to elimination of random accesses to a disk, which is the weakest side of the OS virtual memory. This weakness is especially emphasized in Java applications. A typical JVM prefers allocating new memory blocks to freeing unnecessary ones as soon as possible. This may be very efficient when only physical RAM is in use, but when JVM enters a virtual memory area and a garbage collector tries to recover unused memory blocks from it, the efficiency of a whole application dramatically drops.

References

[1] Vassiliadis P., Simitsis A., Skiadopoulos S., *Modeling ETL Activities asGraphs*. InProc. 4th Intl. Workshop on Design and Management of Data Warehouses, Canada, (2002).

[2] Vassiliadis P., Simitsis A., Georgantas P., Terrovitis M., it A Framework for the Design of ETL Scenarios, CAiSE, (2003).

[3] Galhardas H., Florescu D., Shasha D., Simon E., *Ajax: An Extensible Data Cleaning-Tool*. In Proc. ACM SIGMOD Intl. Conf. On the Management of Data, Teksas, (2000).

[4] Gorawski M., Piekarek M., *Development Environment ETL/JavaBeans*. Studia Informatica, 24 4(56) (2003).

[5] Gorawski M., Sidemak P., *Graphic Design of ETL Applications*. Studia Informatica, 24 4(56) (2003).

[6] Bruckner R., List B., Schiefer J., *Striving Towards Near Real-Time Data Integration for Data Warehouses*. DaWaK, (2002).

[7] Gorawski M., Marks P., *High Efficiency of Hybrid Resumption in Distributed Data Warehouses*. HADIS, (2005).

[8] Gorawski M., Woclaw A., *Evaluation of the Efficiency of Design-Resume/JavaBeans Recovery Algorithm*. Archives of Theoretical and Applied Informatics, 15(1) (2003).