STREAM HANDLING LARGE VOLUME DOCUMENTS IN SITUATIONALLY-ORIENTED DATABASES

V. V. Mironov, A. S. Gusarenko, N. I. Yusupova
Department of Computer Science and Robotics, Ufa State Aviation Technical University, Ufa, Russia
e-mail: mironov@ugatu.su, gusarenko@ugatu.su, yussupova@ugatu.ac.ru

***Abstract**: The article discusses the streaming data processing in large quantities for the situation-oriented databases (SODB). Traditionally SODB provides cached processing of heterogeneous data. This article considered the work with large files that do not fit entirely in memory - containing a number of similar fragments, which can be processed portions. Portions of data are extracted from the input stream, processed in the buffer, and then sent to the output stream. Three variants of implementing this scheme are considered in the framework of the hierarchical situational model of the HSM. The tools offered at the conceptual level are illustrated by the example of processing XML data using PHP software tools, such as XMLReader, XMLWriter, DOM.*

1. Introduction

At the present stage of the information systems development accessible through the network, large data sets accumulated in the course of their life cycle are used in their architecture. As part of such large-scale information systems, databases are used as information support that help to cope with the tasks of processing, storing data and supporting automated functions [1, 2]. Recently, two areas of DBMS development have been identified, the first SQL database and NoSQL, which recently received well-deserved attention by using not only SQL-tools, but also their own data structures in the form of documents suitable for processing and storage, as well as language tools that do not use SQL. Situationally-oriented databases (SODB) is the integrator of diverse (heterogeneous) data. As part SODB formalized dynamic model HSM [3] and programmed functionality to handle the current application state, based on SODB opportunities. In model states, data from relational database tables, such as MySQL [4, 5], are processed to which they are bound. At the initial stage, these were cached processing technologies [6], the technology was based on the DOM (Document Object Model), used in a variety of modern applications that target web applications and other desktop applications. DOM technology used to create a dynamic model states SODB for processing linked to the states of the model data. As soon as the application changed the current state to the state with which the data was associated, dynamic DOM objects were automatically created to process them. With this approach, the application for executing the natural operations of the DBMS for data processing was occupied by the operative memory. In order to avoid the overhead associated with memory utilization, another type of data processing objects based on XMLReader technology was proposed and developed, which was useful for streaming data processing and developing algorithms [6] for processing data in DBMS. Over time, the number of sources for SODB grew into new types of objects [7] have been proposed as a response to, for example, JSON to meet the needs in data processing. The introduction of new types of objects has been further progress in the direction of coverage processing capabilities of heterogeneous data sources that have been considered in the work on SODB [8]. In the next stage of development SODB, when there external sources of heterogeneous data started to develop a web presence in the form of independent SODB services were built into the dynamic model and the remote execution of natural reading query tool for processing data from remote RESTful-service [9, 10]. As the number of sources and the number of data processed increased, it became necessary to organize VDA (Virtual Data Array) [10] structured [11] and invariantly mapped to heterogeneous data stores [12]. In the present conditions, processing in virtual data arrays is required using DPO (Data Processing Objects), such as DOM and Smarty objects [13-15]. The volume of data grows over time, so there are sources [16-19] big data.

2. Stream Handling Large Volume Documents

Custom Data Sources. Basically, this is data received from users who trust and store their data in web applications. Sources of such data can be presented in the form of databases, data stores, individual files and documents in common formats, as well as archives. Such data is also generated by web-based systems for automatic backup, thus creating a dump of information already available in databases, as well as metadata for them.

Web service data. There is another common way of working with sources, that is, not the Web system's own data, but a request for external data distributed on the terms of remote services, for example, by subscription, since Scopus (the international database of metadata and full texts of scientific journals) or Wikipedia, which has its own backup storage - database dumps, it includes its own generated data from the contents of articles and the table of contents and other semantic parts of the encyclopedia. Dumps occupy a significant size for each version, stored in the form of archives, in which are placed article files, with a branched structure, it can be either HTML files or metadata in the form of XML. Such data occupy a large volume even in archives. Wikipedia article files stored together in the archive take up a large amount of data, while the metadata itself, which is a single XML file, occupies a comparable amount of data. Such a dump file can not be normally read in a browser or text editor to view large files, because there is a shortage of memory, there are difficulties in navigating the file.

The task of obtaining and binding data from own sources and remote services. The task of obtaining such data arises if you need to process the associated information of your own source with the data of the remote service. An example can be the user's activity or his work in another service, for example, writing a Wikipedia article or publishing a scientific article in a publication indexed by Scopus. The natural needs to link such data are met by their receipt and processing. When information is received, there are difficulties associated with the large amount of data transmitted over the network, while the download time consumes them.

Processing of received data. When downloading and local data unpacking, the extraction and processing time is doubled, and a significant portion of the external storage will be occupied. To circumvent this kind of difficulty in local processing, so-called wrappers (wrappers) are implemented for streaming reading of deleted data. Thus, data files are not downloaded entirely, but instead are opened on a remote service and read by the program. If the data is not large, it can be read and processed in memory (cached processing) entirely [8], but if they occupy a large part of the memory, the stream processing is used [8]. Stream processing helps to save memory and receive results portion by piece in the right amount.

The problem of obtaining and processing data from a service SODB. The task of processing large data sets when there is a file with a large number of monotonous objects in the tree view of XML, etc. is solved and implemented in modern information systems tools. In most systems, XMLReader's stream processing technology is implemented and used to process large data sets. At the moment, there is no SODB advanced tools can make efficient use of memory to handle the same type of objects contained in the XML-files. The problem of obtaining and processing large amounts of data from the services received for SODB requires decisions by equipping a dynamic model of the specialized stream processing means based on the existing technology of streaming read and write XMLReader and XMLWriter, as well as the possibility of creating a DPO-objects based on the DOM for the intermediate data conversion.

Streams and stream processing. Currently most systems implement programming tools for streaming. The flow regulates the specifics of the data that will be transferred to the application for processing, as well as in the reverse direction with respect to the data that will be output after processing. These streams are oriented to files transmitted over the network, data sent in a compressed form, such as archives. In this case, the flow is meant to be remote resources or their groups, in which the data is stored in a sequential manner, such as an XML file containing sequentially described monotonous objects. Data organized sequentially requires sequential reading by specialized means, for this there are particular cases of implementing such tools XMLReader/XMLWriter, oriented to work with XML. In those conditions, when there are non-standard protocols and data types, it is required to implement additional program code, called a wrapper in the PHP programming system. There are standard wrappers that have their own context, where it is determined which parameters are required to handle a particular known data type. If there are not enough wrappers for non-standard protocols and data types, the programming system is given the opportunity to implement their own wrappers in the form of a separate script, and there are no restrictions on the number of such wrappers. In this paper, we use standard thread wrappers such as php://output that is responsible for the standard output of the PHP system.

3. HSM-stream processing model

Let's consider in general form the idea of the HSM-model for stream processing of documents in the DBMS. The idea is that virtual documents are defined in the model, which, on the one hand, are mapped to the data being processed, and on the other hand, they are associated with the data flow handler. In Fig. 1 summarizes three variants of the organization of the HSM model for streaming data processing: with explicit specification of the thread processing cycle (Fig. 1, a); with the organization of a cycle of processing by means of a loop transition (Fig. 1, b); with implicit specification of the processing cycle (Fig. 1, c). In the `sta:Doc-Stream-Processing` state, two virtual documents (VD) are defined: `doc:Input` and `doc:Output` are the source and destination of the data, respectively. In all three versions of the declaration of virtual documents are the same, the differences are manifested in the organization of the processing of their data. VDs are defined using `doc`-elements the same as in the case of cached processing. The attributes of these elements (for simplicity, not shown in Figure 1) define the physical data store to which the virtual document is displayed. The difference is in setting the VD processing `method` using the method attribute, which indicates the thread's handler used. The `action` attribute opens a stream with certain parameters specifying those portions that will be read from the data source or sent to the receiver. When processing is complete, open threads are closed using the same attribute. The VD processing is performed in the `sub:Proc` submodel, which includes two states: `sta:DocProc` – actual document processing; and `sta:ErrProc` – error handling.

Variant *a* - with an explicit cycle of stream processing. In this variant, stream processing is organized using a `for`-element, intended for cyclic interpretation of the embedded model fragment (`for:EachPiece`, see Fig. 1, *a*). The `scan` attribute here points to VD `doc:Input`, so the `for` element sets the data to be cyclically read from this VD and loads them by default into the buffer of the same name as a `dpo:EachPiece` data object. This object provides a transition to the `sta:ErrProc` state in case of an error (`onLoadErr` attribute) and saving the content in the `doc:Output` document if the `saveDoc` attribute is successfully completed. The actual processing of the contents of the buffer is specified by the nested elements of the `dpo`-element (in Figure 1 are not shown).

Variant *b* - with a loop transition to organize a cycle of stream processing. In this case, the `dpo`-element referencing the VD `doc:Input` (the `srcDoc` attribute) specifies the loading of the next portion of data for processing into the buffer. The processing cycle is provided by the jump element `jmp:DocProc`, which after the buffer processing executes a second transition to the `sta:DocProc` state until the end of the file of the readable document (the `when` attribute) is reached.

Variant *c* - with an implicit definition of the thread processing cycle. In this case, the model fragment related to the data processing looks exactly the same way as for the cached document processing. This implies an implicit loop, such as in case of variant *b*, but without a loopback. Thus, here during the interpretation of the `dpo`-element, there is a call to the input stream handler, loading into the buffer portions of data, processing them and repeating this

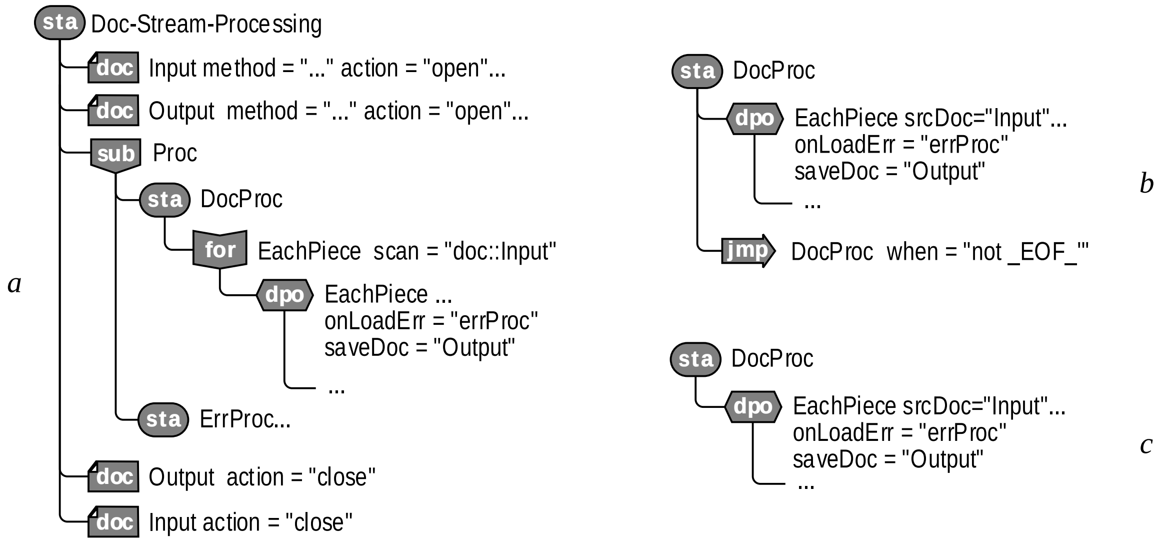
sta Doc-Stream-Processing
doc Input method = "..." action = "open"...
doc Output method = "..." action = "open"...
sub Proc
sta DocProc
for EachPiece scan = "doc::Input"
dpo EachPiece ...
onLoadErr = "errProc"
saveDoc = "Output"
...
sta ErrProc...
doc Output action = "close"
doc Input action = "close"
a
sta DocProc
dpo EachPiece srcDoc="Input"...
onLoadErr = "errProc"
saveDoc = "Output"
...
jmp DocProc when = "not _EOF_"
b
sta DocProc
dpo EachPiece srcDoc="Input"...
onLoadErr = "errProc"
saveDoc = "Output"
...
c

Fig. 1. *General view of the HSM-model of streaming document processing: a - with an explicit cycle; b - with a loop; c - with an implicit cycle*

procedure if the end of the file is not reached.

Comparison of variants. It is currently difficult to say which option is preferable. In variant *a*, the input flow handler used (the request to read the next portion of data) is executed during the interpretation of the `for`-element, in variants *b* and *c* during the interpretation of the `dpo`-element. Variant *c* expands the general concept of interpreting the `dpo`-element, introducing the notion of cyclicity. In this case, the cyclicity of the flow processing is not reflected explicitly in the model. However, in this case, the principle of DPO processing invariance is respected in relation to the definition of VD [5]: the part of the model that specifies the processing of the document remains unchanged when the mapping is changed to physical storage. For example, the processing model does not change externally if, when defining VD, a cached data handler is specified instead of streaming.

4. Example: streaming XML processing

XML-data are "native" to SODB - they used them initially. Let's illustrate the stream processing on this data format. In Fig. Figure 2 shows an example of an HSM model for streaming a large XML document. The XML document used is a file from the archive of the Russian-language Wikipedia. The file size exceeds 100 GB, its cached processing on a normal personal computer is impossible. The structure of the document is shown in Fig. 2, *a*. The `root` element contains the child `siteinfo` (the content is not shown) and a number of child `pages` that reflect information about changes made to Wikipedia articles. Thus, this document refers to a fairly common form of large files containing a large number of small similar fragments. This circumstance makes it possible to effectively organize the stream processing. Let it be required to select from the document those page elements that belong, say, to the author with identifier 121755, and not all information, but only some, in another format is needed. Thus, the processing of the document must include filtering the pages by content, as well as transforming the content. In Fig. 2, *b* shows the corresponding HSM model.

In the `sta:XML-Stream-Processing` state, two virtual multidocuments are specified: `doc:Input`, `ent:RuWiki`, mapped to the XML file of the document being processed, and `doc:Output`, `ent:Tmp`, mapped to the resulting output XML file. With both VD the flow handlers are connected: with the first - XML-Reader - the streaming reader, and with the second - XML-Writer - respectively, the means of streaming XML documents. In the `sub:Proc` sub-model, the virtual documents are streamed according to the scheme of variant a (see Figure 1, a). In the `sta:DocProc` state, the input and output documents are opened, while reading portion of the XML page element, and for the `Output.Tmp` output, the `root` XML element name. The element `for:Page` cyclically reads the XML elements of the page from the input document and places them by default in the DOM-object `dom:Page` of the same name for processing the `Input.RuWiki` input document specifies the `saveDoc` attribute of the dom-element prescribes to save the result in the output document (to send to the output stream) at the end of the processing of the data portion. The actual processing of the data portion is specified by the `jmp:_CONT_` and `rcv:Transform` elements. Using `jmp:_CONT_`, the contents of the page being processed are checked, namely the value of the author's identifier (contributor) using the XPath expression in the when attribute.

XML-data are "native" to SODB - they used them initially. Let's illustrate the stream processing on this data format. In Fig. Figure 1 shows an example of an HSM model for streaming a large XML document. The XML document used is a file from the archive of the Russian-language Wikipedia. The file size exceeds 100 GB, its cached processing on a normal personal computer is impossible. The structure of the document is shown in Fig. 2, *a*. The `root` element contains the child `siteinfo` (the content is not shown) and a number of child `pages` that reflect information about changes made to Wikipedia articles. Thus, this document refers to a fairly common form of large files containing a large number of small similar fragments. This circumstance makes it possible to effectively organize the stream processing. Let it be required to select from the document those page elements that belong, say, to the author with identifier 121755, and not all information, but only

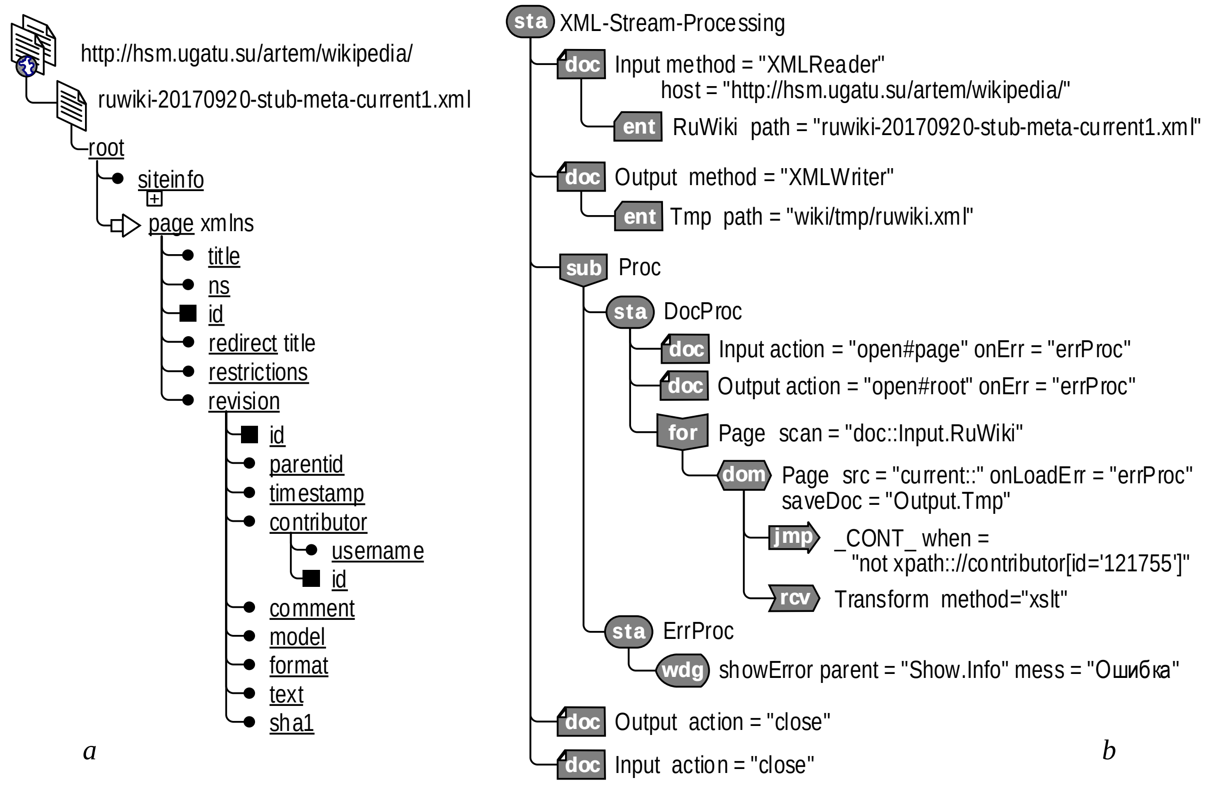
http://hsm.ugatu.su/artem/wikipedia/
ruwiki-20170920-stub-meta-current1.xml
root
siteinfo
page xmlns
title
ns
id
redirect title
restrictions
revision
id
parentid
timestamp
contributor
username
id
comment
model
format
text
sha1
a
sta XML-Stream-Processing
doc Input method = "XMLReader"
host = "http://hsm.ugatu.su/artem/wikipedia/"
ent RuWiki path = "ruwiki-20170920-stub-meta-current1.xml"
doc Output method = "XMLWriter"
ent Tmp path = "wiki/tmp/ruwiki.xml"
sub Proc
sta DocProc
doc Input action = "open#page" onErr = "errProc"
doc Output action = "open#root" onErr = "errProc"
for Page scan = "doc::Input.RuWiki"
dom Page src = "current::" onLoadErr = "errProc"
saveDoc = "Output.Tmp"
jmp _CONT_ when =
"not xpath::/contributor[id='121755']"
rcv Transform method="xslt"
sta ErrProc
wdg showError parent = "Show.Info" mess = "Ошибка"
doc Output action = "close"
doc Input action = "close"
b

Fig. 2. *An example of an HSM model for streaming XML data:*
a - the structure of the XML document; b - stream processing model

some, in another format is needed. Thus, the processing of the document must include filtering the pages by content, as well as transforming the content. In Fig. 2, *b* shows the corresponding HSM model. In the `sta:XML-Stream-Processing` state, two virtual multidocuments are specified: `doc:Input`, `ent:RuWiki`, mapped to the XML file of the document being processed, and `doc:Output`, `ent:Tmp`, mapped to the resulting output XML file. With both VD the flow handlers are connected: with the first - XML-Reader - the streaming reader, and with the second - XML-Writer - respectively, the means of streaming XML documents. In the `sub:Proc` sub-model, the virtual documents are streamed according to the scheme of variant a (see Figure 1, a). In the `sta:DocProc` state, the input and output documents are opened, while reading portion of the XML page element, and for the `Output.Tmp` output, the `root` XML element name. The element `for:Page` cyclically reads the XML elements of the page from the input document and places them by default in the DOM-object `dom:Page` of the same name for processing the `Input.RuWiki` input document specifies the the `saveDoc` attribute of the dom-element prescribes to save the result in the output document (to send to the output stream) at the end of the processing of the data portion. The actual processing of the data portion is specified by the `jmp:_CONT_` and `rcv:Transform` elements. Using `jmp:_CONT_`, the contents of the page being processed are checked, namely the value of the author's identifier (contributor) using the XPath expression in the `when` attribute. XML-data are "native" to SODB - they used them initially. Let's illustrate the stream processing on this data format. In Fig. Figure 1 shows an example of an HSM model for streaming a large XML document. The XML document used is a file from the archive of the Russian-language Wikipedia. The file size exceeds 100 GB, its cached processing on a normal personal computer is impossible. The structure of the document is shown in Fig. 2, *a*. The `root` element contains the child `siteinfo` (the content is not shown) and a number of child `pages` that reflect information about changes made to Wikipedia articles. Thus, this document refers to a fairly common form of large files containing a large number of small similar fragments. This circumstance makes it possible to effectively organize the stream processing. Let it be required to select from the document those page elements that belong, say, to the author with identifier 121755, and not all information, but only

some, in another format is needed. Thus, the processing of the document must include filtering the pages by content, as well as transforming the content. In Fig. 2, *b* shows the corresponding HSM model. In the `sta:XML-Stream-Processing` state, two virtual multidocuments are specified: `doc:Input`, `ent:RuWiki`, mapped to the XML file of the document being processed, and `doc:Output`, `ent:Tmp`, mapped to the resulting output XML file. With both VD the flow handlers are connected: with the first - XML-Reader - the streaming reader, and with the second - XML-Writer - respectively, the means of streaming XML documents. In the `sub:Proc` sub-model, the virtual documents are streamed according to the scheme of variant a (see Figure 1, a). In the `sta:DocProc` state, the input and output documents are opened, while reading portion of the XML page element, and for the `Output.Tmp` output, the `root` XML element name. The element `for:Page` cyclically reads the XML elements of the page from the input document and places them by default in the DOM-object `dom:Page` of the same name for processing the `Input.RuWiki` input document specifies the the `saveDoc` attribute of the dom-element prescribes to save the result in the output document (to send to the output stream) at the end of the processing of the data portion. The actual processing of the data portion is specified by the `jmp:_CONT_` and `rcv:Transform` elements. Using `jmp:_CONT_`, the contents of the page being processed are checked, namely the value of the author's identifier (contributor) using the XPath expression in the when attribute. If the author's identifier does not match the specified value, the processing is interrupted, thereby this `page` element is ignored in the output stream. Using the `rcv:Transform` receiver, XSL transforms the contents of the DOM object before sending it to the output stream. XSLT is a powerful tool for converting XML data based on style sheets (Stylesheets), both in XML and in other formats. The conversion method is specified by the `method` attribute, by default the style sheet, which is the same as the element, is used, i.e., `Transform.xsl`. The details of the transformation are omitted here for brevity. When errors occur, the state goes to `sta:ErrProc`, where the corresponding message is generated to the user.

5. Conclusion

In the article the questions of the task of streaming data processing in DBMS on the basis of declarative HSM-models of integration of heterogeneous data sources are considered. For data processing in the HSM, it is provided, firstly, to set VD - virtual documents displayed on the physical storage, and secondly, to set DPO - data processing objects, into which data from VD is loaded. The previously provided HSM capabilities are focused on cached processing, in which VD is completely loaded into the main memory. This imposes restrictions on the amount of processed documents. For processing large data, computing platforms provide streaming processing capabilities, in which data is downloaded in batches from memory to the memory from the source, processed there, and uploaded to the data receiver. The task is to develop declarative means of specifying stream processing within the framework of the HSM. The introduction of stream processing required additional provision in the HSM:

1) The ability to specify a particular streaming handler when specifying VD, and specifying the portions of the data extracted from the physical store or placed in the peg. For this, the method attribute is provided in the virtual document definition;

2) Possibility of setting cyclic processing of data portions VD. For this, three organization variants are proposed: with an explicit cycle, which is provided by the `for`-element; with the cycle provided by the `jmp`-element; with an implicit cycle provided by a modified `dpo`-element.

The possibilities of specifying stream processing are demonstrated in the example of processing a large XML document. Portions of the document are read by the XMLReader tool, loaded into the DOM-object and processed in it, and then output to the output stream by the XMLWriter tool. Thus, the general concept of VD / DPO, used in DBMS for cached processing, with minor modifications is also applicable for the stream processing of large documents.

Acknowledgments

This work is supported by RFBR grant № 16-07-00239.

References

1. Pokorny J., “NoSQL databases: a step to database scalability in web environment”, *International Journal of Web Information Systems*, 2013; 9(1):69–82.

2. Mironov V. V., Gusarenko A. S., Dimetriev R. R., Sarvarov M. R. “The Personalized Documents Generating Using DOM-objects in Situation-Oriented Databases”, *Vestnik UGATU*. 2014; 65:191–197. (In Russian).

3. Kosar T., Bohra S., Mernik M. Domain-Specific Languages: A Systematic Mapping Study // *Information and Software Technology*. 2016; 71:77–91.

4. Gusarenko A. S. “Improvement of Situation-Oriented Database Model for Interaction with MYSQL”, *Izvestiya vysshikh uchebnykh zavedeniy. Priborostroenie.* [Journal of Instrument Engineering] 2016; 59:355–363. (In Russian).

5. Mironov V. V., Gusarenko A. S., Yusupova N. I., “Displaying virtual the XML-documents to MySQL tables in situation-oriented databases, “distributed approach””, *Informacionnye*

tehnologii i vychislitel'nye sistemy [Information technology and computer systems], 2017; 1:77–89. (In Russian).

6. Mironov V. V., Gusarenko A. S., Yusupova N. I. “Situation-oriented databases: current state and prospects for research”, *Vestnik UGATU*. 2015; 68:188–199. (In Russian).
7. Gusarenko A. S., Mironov V. V. “Smarty-objects: Use Case of Heterogeneous Sources in Situationally-Oriented Databases”, *Vestnik UGATU*. 2014; 64:242–252. (In Russian).
8. Gusarenko A. S., Mironov V. V. “Heterogeneous Document Sources in Situationally-Oriented Databases”, *Vestnik UGATU*. 2015; 19:124–131. (In Russian).
9. Mironov V. V., Gusarenko A. S. “Using of RESTful-Services in Situationally-Oriented Databases”, *Vestnik UGATU*. 2015; 67:232–239. (In Russian).
10. Mironov V. V., Gusarenko A. S., Yusupova N. I. “Situation-Oriented Database: Integration of XML Data and Relational Environment” // *Sistemy upravlenija i informacionnye tehnologii* [Automation and Remote Control]. 2016; 65:48–56. (In Russian).
11. Mironov V. V., Gusarenko A. S., Yusupova N. I. “Structuring Virtual Multi-Documents in Situationally-Oriented Databases by Means of Entry-Elements”, *Trudy SPIIRAN* [SPIIRAS Proceedings]. 2017; 53:225–243. (In Russian).
12. Mironov V. V., Gusarenko A. S., Yusupova N. I., “The Invariance of The Virtual Data in The Situationally Oriented Database When Displayed on Heterogeneous Data Storages”, *Vestnik komp'iuternykh i informatsionnykh tekhnologii* [Herald of Computer and Information Technologies], 2017; 151:29–36. (In Russian).
13. Osvaldo S. S. Jr. etc. “Developing software systems to Big Data platform based on MapReduce model: An approach based on Model Driven Engineering”, *Information and Software Technology*. 2017; 92:30–48.
14. Cobo M. J., López-Herrera A.G., Herrera-Viedma E. “A relational database model for science mapping analysis”, *Acta Polytechnica Hungarica*. 2015; 6:43–62.
15. Arevalo C. etc. “A metamodel to integrate business processes time perspective in BPMN 2.0”, *Information and Software Technology*. 2016; 77:17–33.
16. Amanatidis T., Chatzigeorgiou A. “Studying the evolution of PHP web applications”, *Information and Software Technology*. 2016; 72:48–67.
17. Agh H., Ramsin R. “A pattern-based model-driven approach for situational method engineering”, *Information and Software Technology*. 2016; 78:95–120.
18. Mironov V. V., Gusarenko A. S., Yusupova N. I. Situation-oriented databases: document management on the base of embedded dynamic model / CEUR Workshop Proceedings (CEUR-WS.org): Selected Papers of the XI International Scientific-Practical Conference Modern Information Technologies and IT-Education (SITITO 2016), Moscow, Russia, November 25-26, 2016. P. 238–247.
19. Djukic V., Lukovic I., Popovic A., Ivancevic V., “Model Execution: An Approach based on extending Domain-Specific Modeling with Action Reports”, *Computer Science and Information Systems*, 2013; 4:1585–1620.