ATLANTIS PRESS

Advances in Intelligent Systems Research, volume 166

7th Scientific Conference on Information Technologies for Intelligent Decision Making Support (ITIDS 2019)

Stream Documents Processing Invariance in Situation-Oriented Databases

Valeriy Mironov
Computer Science & Robotics Dept
Ufa State Aviation Technical University
Ufa, Russia
mironov@ugatu.su

Artem Gusarenko
Computer Science & Robotics Dept
Ufa State Aviation Technical University
Ufa, Russia
gusarenko@ugatu.su

Nafisa Yusupova
Computer Science & Robotics Dept
Ufa State Aviation Technical University
Ufa, Russia
yussupova@ugatu.ac.ru

***Abstract* — Data stream processing in situational-oriented databases (SODB) is discussed. SODB is a heterogeneous data integrator operating under the control of the built-in hierarchical situational model (HSM). The processed data is defined in the HSM as virtual documents (VD) mapped to real data stores. HSM specifies the loading of a VD data into Data Processing Objects (DPO), the data transformation in DPO, and unloading of processing results to a VD. Stream processing of large documents that do not fit entirely into RAM is discussed. A VD model involving multiple pieces of data that can be processed separately is considered. Portions of data are extracted from the VD input stream, processed in the DPO, and then sent to the VD output stream. Invariance should ensure the independence of the DPO model when the VD model changes. Invariance to different schemes streaming data is discussed. Algorithms for loading and unloading DPO are considered, ensuring invariance due to the hidden cyclical processing of data portions. The proposed solutions are illustrated with examples of processing XML and JSON documents.**

Keywords — situation-oriented database, built-in dynamic model, streaming data processing, DOM, HSM, XML, JSON.

I. INTRODUCTION

The Polyglot Persistence. The phenomenon of "big data" generated the need for efficient processing of heterogeneous information in large documents within a single web application [1]. For example, ETL (Extract, Transform, Load) processes in business intelligence systems, as a rule, loads the data warehouse from a variety of heterogeneous sources: file systems, relational databases, NoSQL databases, web services and other [2]. Therefore, an application needs an integration layer that provides interaction with heterogeneous data sources [3].

The SODB. Situation-oriented databases (SODB) are developed and researched by the authors as tools for integrating heterogeneous data at a high level of abstraction [4]. The SODB implements the Model-Driven Approach: hierarchical situational model of the HSM, built in SODB, is the basis for its functioning. HSM is a hierarchy of submodels and is processed by a special type of interpreter — Finite State Machine. In HSM, a submodel specifies a set of states. For each state, the following conditions can be specified: (1) jumps to other states of the submodel, (2) dives to internal submodels, (3) actions performed in this state. The HSM interpreter (HSMI) periodically performs a hierarchical traversal of the current submodels, changes their current states, performs associated actions. The current HSM states are retained between interpretation cycles.

The Data Processing. Special actions specifying data processing may be set in the states of the HSM. The concept of virtual documents [5] provides for the separation of two aspects: (1) a description of the mapping of a virtual document (VD) to the actual data storage; (2) the task of processing VD with the help of data processing facilities of the DPO. The VD specifications allow the interpreter to access a specific type of external storage using appropriate tools. DPO specifications set the necessary actions for the VD. Note that cached data processing was initially assumed in SOBD: The VDs extracted from the external store should have been completely loaded into the DPO.

The Invariance Principle. Difficulties in the organization of the virtual documents processing may occur when changing the VD mapping to real data [5]. During development or operation, the way in which real data is stored may change. In turn, this will entail a change in the VDs mapping specifications. Will this require a change in VDs processing specifications? The introduced invariance principle states: a change in the physical location of the VD data should not be accompanied by changes in the DPO specifications for processing this VD. In [4], invariance assurance is illustrated for cached processing of VD in XML format when mapped on documents in files and relational database [6].

The Stream Processing. In the case of "big data", the VD may not fit entirely into the computer's RAM, which prevents the cached processing in the SODB. For such cases, software systems provide the capabilities and tools for streaming data input and output. This allows data to be processed in foreseeable chunks (portions) as they arrive from the data source, and to send chunks of processed data to the receiver as soon as it is ready. The organization of streaming data processing in SODB was considered in [7] within the framework of the concept of separation of VD and DPO. It is shown that the introduction of stream processing may be accompanied by a violation of the invariance principle. That is, the transition from cached processing to streaming will require changes to the DPO specifications along with changes to the VD specifications.

The ideas for providing data stream processing in the SODB, considering the principle of invariance, are discussed in this paper.

II. THE DATA STREAM MODEL

The data processed by the stream at the lower level of abstraction is a sequence of bytes. In general, data processing should include the extraction of logical fragments from the data stream for further processing.

We are considering a higher-level abstraction data stream that takes into account a data format, such as XML [8,9], JSON CSV, or some other. These formats define logical pieces of data and their internal structure. Thus, the XML format assumes hierarchical markup of text using opening and closing tags. JSON [10] uses markup using paired braces and square brackets. The CSV format provides for the division of data into separate records, as well as the division of records into separate fields.

Thus, it is assumed that it is possible to isolate on the fly logical data fragments [11] (portions) of data from the data stream necessary for processing. The parameters required for such isolation are specified in the specifications of virtual documents. In addition, it is assumed that these portions of data can be cached in RAM. That is, "big data" [12-15] is considered, which is a large set of similar elements. Moreover, each element has a foreseeable size that allows cached processing.

The considered data model for XML, JSON and CSV formats is presented in Fig. 1.

XML format (Fig 1, *a*). In accordance with the XML format [16], the `XMLDoc` document begins with service information (processing instructions, comments), represented in the form of the `Title` aggregate. This is followed by the root element, indicated in the figure as `DocElement`. This is a container for document content, tagged with `<tagName>`*content*`</tagName>`. The root element may contain `DocAttr` attributes related to the document. Top level elements nested as `TopElement` are nested inside the root element. The document may contain several top-level elements (very much in our case). Top-level elements have content defined by its attributes and internal (hierarchical) markup (omitted from the figure). It is assumed that the processing of the document consists in the sequential processing of the top-level elements of `TopElement`. A variant of this may be the sequential processing of elements nested in the top-level elements.

JSON format (Fig 1, *b*). In accordance with the rules of the JSON format [17], the `JSONDoc` document is the root element, denoted in the figure as a `DocElement`. The root element can be an `array` enclosed in square brackets, or an `object` enclosed in braces. In the first case, the element contains a list of `values`, and in the second, a list of pairs `key-value`. *Values*, in turn, can be atomic or complex: `arrays` or `objects`. It is assumed that the processing of the document consists in the sequential processing of the list contained in the `DocElement` element. A variant of this may be the sequential processing of lists nested in the elements of the top-level list.

CSV format (Fig 1, *c*). In accordance with the rules of the CSV format [18], the `CSVDoc` document is a set of records, delimited by a line break. Records consist of fields, separated by commas. The first entry may be a header (designated in the figure as `HeaderRec`). The header contains information related to the entire document. Typically, the header contains the document field names. The main part of the document includes many entries in the form of a list of field values (denoted as `DocRec`). It is assumed that the processing of the document is the sequential processing of records.

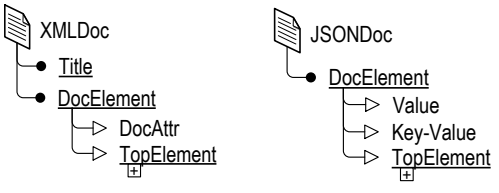
Programming languages provide a variety of tools that allow you to analyze data flow on the fly, detecting markup characters, line break characters, and isolating logical portions of data. For example, standard tools such as `SAX`, `XMLReader`, `XMLWriter`, etc. are offered for streaming processing of large XML documents [19] on the PHP platform. For JSON and CSV documents there are similar solutions from third-party developers.

Caching vs Streaming. With cached processing, the entire document is loaded into the DPO and on this basis the result is generated — the output document. When streaming [19] in DPO, the next logical portion of the data is loaded, that is, DPO is used as an intermediate buffer. The result of the intermediate processing is sent to the output stream.

Portion processing of data from the stream. A portion of the data loaded into the DPO for processing may include several logical elements that are processed independently. That is, you can process several XML elements, JSON elements, CSV records at a time. This can speed up the process in some cases. In the case of the processed portion may contain a single element.

Model tool to read or write stream:

- `Open` — open stream to read or write data.
- `PrepareFirst` — prepare parameters for getting or putting the first portion of stream data.
- `GetFirst` / `PutFirst` — get from stream or put to stream first portion of data.
- `PrepareNext` — prepare parameters for getting or putting the next portion of stream data.
- `GetNext` / `PutNext` — get from stream or put to the stream the next portion of data.
- `Close` — close data stream.



a *b* *c*

Fig. 1. Data Models: XML (*a*), JSON (*b*), CSV (*c*).

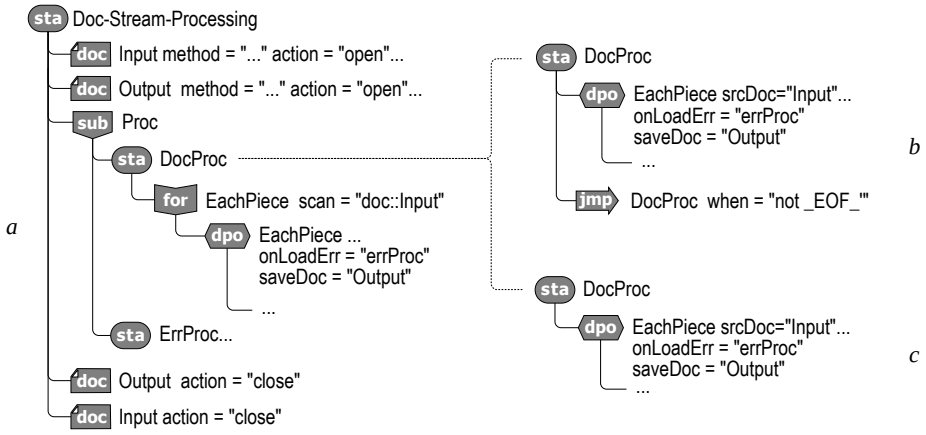


Fig. 2. Variants for specifying stream processing in the HSM model: *a* – explicit loop; *b* – loop jump; *c* – implicit loop.

Virtual document model. In the definition of a virtual document that is processed as a data stream, thus, the following information may be specified:

- Address and parameters of the source or receiver of data (for example, web service address, access parameters).
- A tool to read or write data (for example, XMLReader or XMLWriter).
- Parameters of reading or writing data (for example, an XPath expression that addresses processed elements in an XML document that is read from a stream).
- Portion size, i.e. the maximum number of elements read from a stream or written to the stream at a time within the same portion to be processed.

III. PROCESSING OF DATA FLOW

In Fig. 2 the three variants for organizing the HSM model for stream processing, proposed in [20], are summarizes: (a) based on the explicit assignment of the stream processing loop; (b) based on jump loop processing; (c) based on an implicit processing loop.

In the `sta:Doc-Stream-Processing` state, two virtual documents (VD) are specified: `doc:Input` (data source) and `doc:Output` (data receiver). Virtual document declarations are the same in all three variants, the differences are manifested in the organization of the processing of their data.

Virtual documents, as in the case of cached processing, are defined using `doc` -elements. The attributes of these elements (not shown in Fig. 1 for simplicity) determine the physical data storage to which the virtual document is mapped. The difference lies in defining the processing method using the `method` attribute, which specifies the stream handler to use. The `action` attribute opens a stream with certain parameters defining those portions that will be read from the data source or sent to the receiver. Closing open streams upon completion of processing is also provided by the `action` attribute.

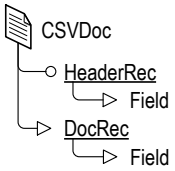
Virtual document processing is performed in `sub:Proc` submodels. The `sta:DocProc` state sets the actual document handling, and the `sta:ErrProc` state sets the error handling.

Variant A — explicit streaming processing loop. In this variant, stream processing is organized using a `for`-element intended for cyclic interpreting the model fragment embedded in it (`for:EachPiece`, see Fig. 1, *a*). The `scan` attribute here references to a virtual document `doc:Input`. The `for` element specifies a cyclic reading of data portions from this document and loading into a data processing object `dpo:EachPiece`. This object provides a jump to the state `sta:ErrProc` in case of an error (attribute `onLoadErr`) and saving content to document `doc:Output` in case of successful completion of processing (attribute `saveDoc`). Processing the contents of the buffer is specified by nested elements of the `dpo`-element (not shown in Fig. 1).

Variant b — loopback jump for streaming loop processing. In this case, the `dpo`-element referring to `doc:Input` (`srcDoc` attribute) sets the loading of the next portion of data for processing into the buffer. The processing cycle is organized using the `jmp:DocProc` element, which, after processing the buffer, performs a repeated jump to the `sta:DocProc` state until it reaches the end of the readable document file (the `when` attribute).

Variant c — implicit loop streaming processing. In this case, the fragment of the model relating to data processing looks outwardly the same as for the case of cached document processing. This implies an implicit loop, such as in the case of variant *b*, but without a loopback. Thus, during the interpretation of the `dpo`-element, the input stream handler is accessed, portions of data are loaded into the buffer, processed and repeated, if the end of the file is not reached.

Comparison of variants. In variant *a*, the input flow control (request to read the next piece of data) is performed during the interpretation of the `for`-element. In variants *b* and *c*, the



control is performed during the interpretation of the `dpo`-element. Variant *c* extends the overall concept of interpreting

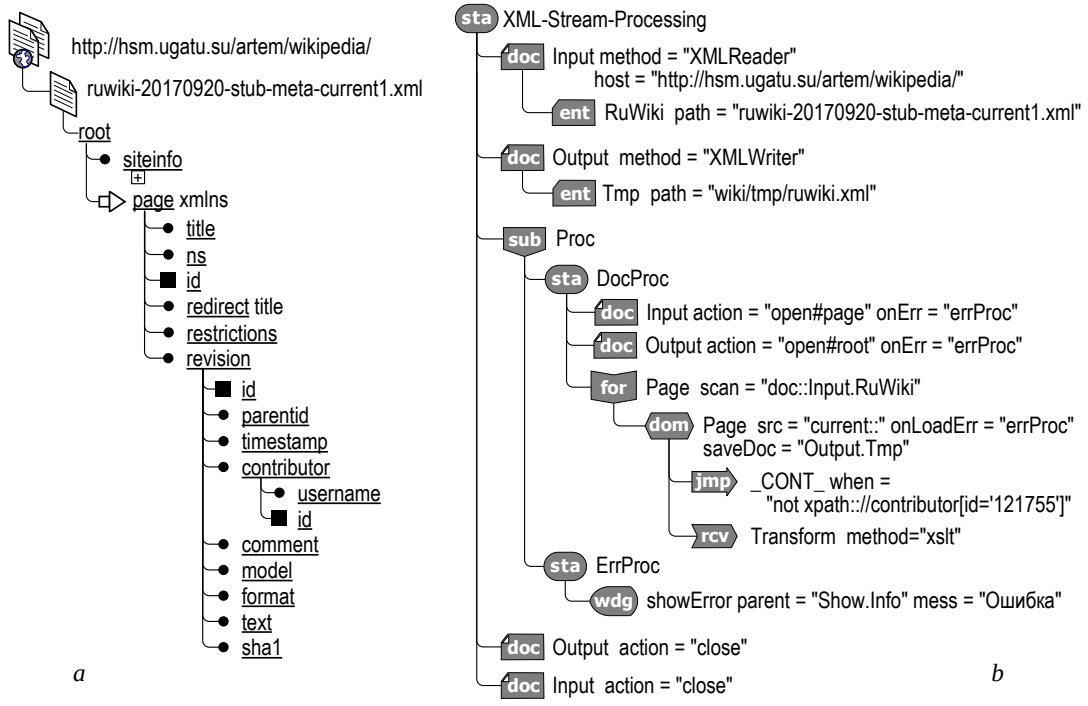


Fig. 3. An example of streaming XML processing: *a* — XML document structure; *b* — HSM streaming processing model.

the `dpo`-element, introducing a mechanism of cyclicity. In this case, the cyclical nature of stream processing is not explicitly reflected in the model [21-24]. However, in this case, the principle of invariance of the DPO processing with respect to the definition of VD is partially observed, i.e. that part of the model that specifies the processing of the document remains unchanged when the mapping to the physical store changes.

For example, the processing model will change little in appearance if a cached data handler is specified when defining a virtual document instead of a streaming document.

IV. EXAMPLES

A. *Streaming XML Processing*

Initially, SODB was focused on XML data, i.e., XML data is native to SODB [25]. Let's illustrate stream processing in this data format. In Fig. 3 an example of an HSM model for streaming a large XML document is shown. The XML document used is a file from the Russian-language Wikipedia archive. The file size exceeds 100 GB, and therefore its cached processing on a regular personal computer is difficult.

The structure of the document is shown in Fig. 3, *a*. The `root` element contains a child `siteinfo` element (content not shown) and many `page` child elements reflecting information about changes made to Wikipedia articles. Thus, this document refers to a common type of large files containing many small, single-type fragments. This circumstance allows you to effectively organize stream processing.

Suppose you want to select from the document those `page` elements that belong to the author with the identifier 121755. At the same time, not all the information is needed, but only some of it, and in a different format. Thus, document processing should include filtering pages based on their content, as well as content transformation.

In Fig. 3, *b* shows the HSM model. In `sta:XML-Stream-Processing` state, two virtual multi-documents are specified: (1) `doc:Input / ent:RuWiki`, which is displayed on the XML file of the document to be processed, and (2) `doc:Output / ent:Tmp`, which is displayed on the resulting output XML file. Stream handlers are associated with virtual documents. XMLReader is a streaming reading tool. XMLWriter is a tool for streaming writing XML documents.

In the submodel `sub:Proc`, stream processing of virtual documents is performed according to the variant *a* (see Fig. 2, *a*). In the `sta:DocProc` state, the input and output documents are opened. The input document `Input.RuWiki`, the XML element `page` is set as a portion of data reading. For the output document `Output.Tmp`, the root XML element name is set as `root`. The `for:Page` element provides looping reads from the input document `page` elements and places them in the `dom:Page` DOM object for processing. The `dom` element's `saveDoc` attribute instructs to save the result in the output document (send to the output stream) after the processing of the piece of data is completed.

The processing of a piece of data is controlled by the elements `jmp:_CONT_` and `rcv:Transform`. Using the `jmp:_CONT_` jump, the contents of the `page` element being processed are checked, that is, the value of the contributor identifier is checked according to the XPath expression in the `when` attribute. If the author identifier does not match the specified value, processing is interrupted, that is, this `page` element is ignored in the output stream. Using

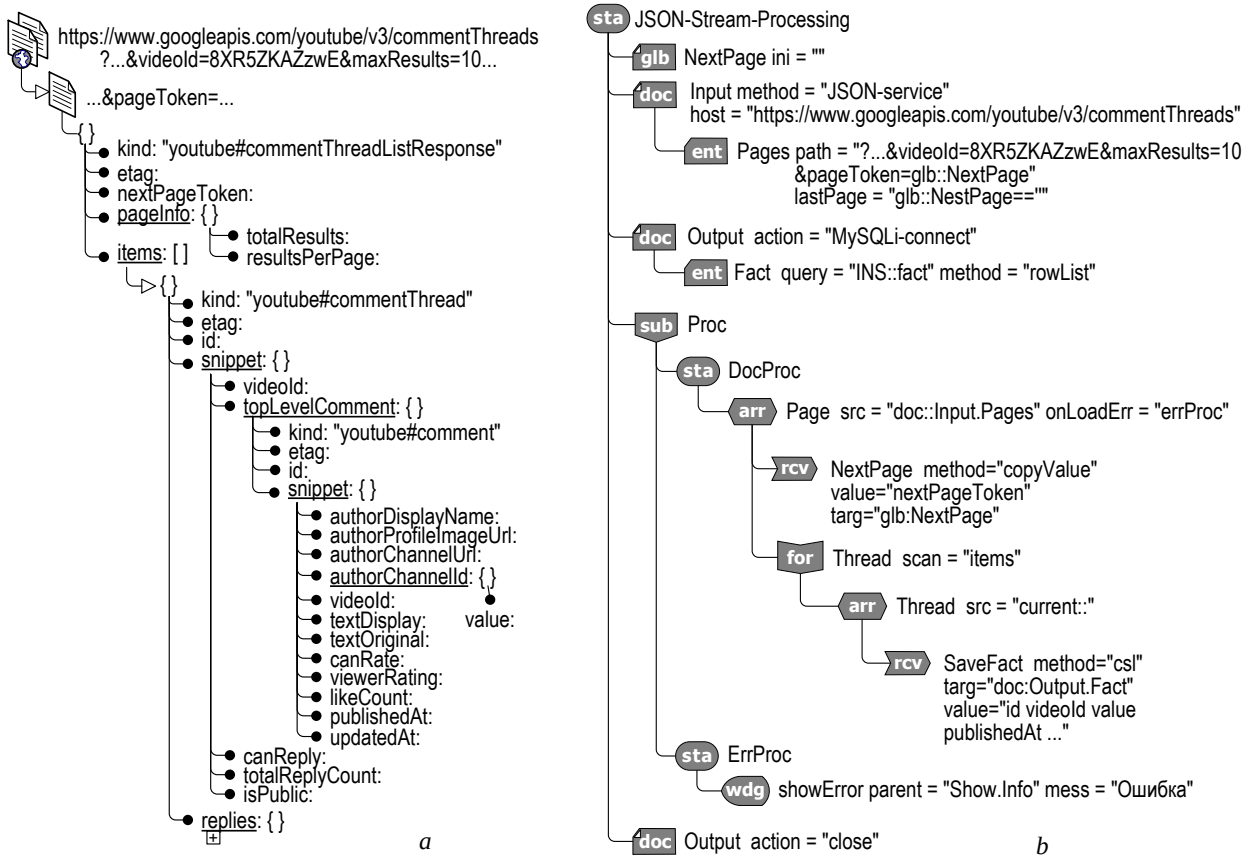


Fig. 4. Example of stream processing of JSON data: *a* — JSON document structure; *b* — HSM streaming processing model

the `rcv:Transform` receiver, an XSL transformation is performed on the contents of the DOM object before sending it to the output stream. XSLT is a powerful tool for transforming XML data based on style sheets. The transformation method is specified by the `method` attribute; The default style sheet is `Transform.xsl`. Details of the conversion are omitted here for brevity.

When errors occur, a jump to the `sta:ErrProc` state occurs, where a corresponding message is generated to the user.

B. Stream processing JSON

Data in JSON format is often used by web services now. We illustrate stream processing on this data format. In Fig. 4 an example of an HSM model [26] for stream processing a JSON document is shown.

The JSON document used contains information about the comments on a video posted on the YouTube video-sharing website. Documents of this type are provided through the YouTube Data API. The structure of the document is shown in Fig. 4, *a*. The web service `commentThreads` provides comments to the selected video as a series of pages. Each page has a local identifier `pageToken` and contains a limited number of top-level comment threads. Each loaded page is a JSON object containing information in the form of key–value pairs. The nested JSON array of `items` contains a set of JSON objects, each of which reflects data [27] about a single top-level comment thread.

Suppose you want to get comments to some video, for example, video with the identifier `8XR5ZKAZzwE`. For each of the top-level comments, you need to extract certain parameters, such as comment identifier, video identifier, channel author identifier, comment publication time, etc. The extracted parameters must be inserted into the fact table of the relational database as a comment entry. The fact table is further used for analysis purposes.

In Fig. 4, *b* the corresponding HSM model is shown. In the `sta:JSON-Stream-Processing` state, the variable `glb:NextPage` and two virtual multi-documents are set: (1) `doc:Input` / `ent:Pages`, which is mapped to a web service that supplies a JSON document, and (2) `doc:Output` / `ent:Fact`, which is mapped on the `Fact` table of the MySQL relational database in insert mode [18–27]. The following parameters were set for the input document: video identifier; maximum number of comments per page, etc. The variable `glb:NextPage` is used to control the loading of pages. For this, a reference to `glb:NextPage` is injected into the list of parameters in the `ent:Pages` element (parameter `pageToken`). The `lastPage` attribute sets the condition for finding the last page of the input document.

In the submodel `sub:Proc`, stream processing of virtual documents is performed according to the variant *c* (see Fig. 2, *c*). As in the previous example, in the `sta:DocProc` state, the input virtual document is processed and the output virtual document is formed. The state `sta:ErrProc` is for error handling; it forms the corresponding message to the user.

The next JSON page is read from the input virtual document and loaded into the DPO object `arr:Page` for processing. This object is an associative array, well combined with data in JSON format [12]. The `onLoadErr` attribute instructs the transition to the `sta:ErrProc` state in the event of a load error.

Processing the loaded page is defined by three elements. Using the first element (`rcv:NextPage`), the `pageToken` value for the next page of the stream is written to variable `glb:NextPage`. Using the second element (`for:Thread`), looping the elements of the `items` JSON array as a subarray `arr:Thread` is performed. Using the third element (`rcv:SaveFact`) a fact table row is formed and written to a relational database. Details of the transformation are omitted here for brevity. After processing all the internal elements of the `arr:Page` DPO object, the `lastPage` condition is checked (empty value of variable `glb:NextPage`). If the condition is not true, then the next page of the stream is loaded and processed.

V. Conclusions

Stream processing in situational databases is based on the separate description of virtual documents and data processing objects in a situational hierarchical model. Virtual documents set the mapping to the physical data store. And data processing objects specify the processing of data loaded from virtual documents. When streaming, large data is loaded into data processing objects in portions of limited size. Thus, the extension of the approach used in cached processing, when the document is completely loaded into RAM, is used. This is generally consistent with the principle of invariance, according to which the part of the model that specifies the processing of a document must be invariable when the mapping changes to physical store.

However, it is not possible to fully ensure invariance when moving from cached processing to streaming. During stream processing, it is not possible to make completely independent the declarations of the virtual document mapping and the declarations for virtual document processing. This is because when defining a virtual document, information is needed that becomes available only when processing the next portions of the data stream. So, in the specifications of a virtual document, you need to specify the condition for the end of the data stream, which can only be determined during the processing of the last portion of the stream.

The capabilities of the stream processing task are demonstrated by the example of processing a large XML document. Portions of the document are read by the XMLReader tool, loaded into a DOM object, processed in it, and then output to the output stream with the XMLWriter tool. Another considered example illustrates the processing of a large JSON document, which is generated page by page by the YouTube service.

ACKNOWLEDGMENT

Russian Foundation for Basic Research grant No 19-07-00682

References

[1] Amanatidis T., Chatzigeorgiou A. Studying the evolution of PHP web applications In: *Information and Software Technology*. 2016. V. 72. P. 48–67.

[2] Sadalage P. J., Fowler M. NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, 2013.

[3] Gusarenko A. S., Mironov V. V. Heterogeneous Document Sources in Situationally-Oriented Databases In: *Vestnik UGATU*. 2015. V.19. No. 4. P. 124–131. (In Russian).

[4] Gusarenko A.S. Model for creating documents in Office Open XML format based on situationally-oriented databases In: *Applied Informatics*. 2015. V. 10. No. 3 (57). P. 63–76. (In Russian).

[5] Mironov V. V., Gusarenko A. S., Dimetriev R. R., Sarvarov M. R. The Personalized Documents Generating Using DOM-objects in Situation-Oriented Databases In: *Vestnik UGATU*. 2014. V.18. No. 4 (65). P. 191–197. (In Russian).

[6] Gusarenko A.S. Improvement of Situation-Oriented Database Model for Interaction with Mysql In: *Journal of Instrument Egineering*. 2016. V. 59. No. 5. P. 355–363. (In Russian).

[7] Kosar T., Bohra S., Mernik M. Domain-Specific Languages: A Systematic Mapping Study In: *Information and Software Technology*. 2016. V. 71. P. 77–91.

[8] Mironov V.V., Gusarenko A.S., Yusupova N.I. Displaying virtual XML-documents on MySQL tables in the situation-oriented databases, "distributed approach" In: *Journal of Information Technologies and Computing Systems*. 2017. No. 1. P. 77–89. (In Russian).

[9] Mironov V. V., Gusarenko A. S., Yusupova N. I. Situation-Oriented Databases: Current State and Prospects for Research In: *Vestnik UGATU*. 2015. V.19. No. 2 (68). P. 188–199. (In Russian).

[10] Gusarenko A. S., Mironov V. V. Smarty-objects: Use Case of Heterogeneous Sources in Situation-Oriented Databases In: *Vestnik UGATU*. 2014. V.18. No. 3(64). P. 242–252. (In Russian).

[11] Mironov V. V., Gusarenko A. S. Using of RESTful-Services in Situationally-Oriented Databases In: *Vestnik UGATU*. 2015. V.19. No. 1 (67). P. 232–239. (In Russian).

[12] Mironov V. V., Gusarenko A. S. Yusupova N.I. Integration of Virtual Multidocument Mappings into Real Data Sources in Situational-Oriented Databases In: *Applied Informatics*. 2018. V. 13. No. 3(75). P. 47–60. (In Russian).

[13] Mironov V.V., Gusarenko A.S., Yusupova N.I. Structuring virtual multi-documents in situationally-oriented databases by means of entry-elements In: *SPIIRAS Proceedings*. 2017. V. 4. No. 53. P. 225–242. (In Russian).

[14] 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 In: *Herald of Computer and Information Technologies*. 2017. No. 1(151). P. 29–36. (In Russian).

[15] Osvaldo S. S. Jr. и др. Developing software systems to Big Data platform based on MapReduce model: An approach based on Model Driven Engineering In: Information and Software Technology. 2017. V. 92. P. 30–48.

[16] Extensible Markup Language (XML). [Online]. Available: https://www.w3.org/XML/. [Accessed: 09-Apr-2019].

[17] JSON. [Online]. Available: https://www.json.org/. [Accessed: 09-Apr-2019].

[18] Common Format and MIME Type for Comma-Separated Values (CSV) Files. [Online]. Available: https://tools.ietf.org/html/rfc4180. [Accessed: 09-Apr-2019].

[19] Gusarenko A.S., Mironov V.V., Yusupova N.I. Stream processing of large documents in situational databases In: *Proceedings of the 6-th International Conference Information Technologies for Intelligent Decision Making Support*. Ufa, Russia: Ugatu, 2018. P. 7–12.

[20] V. V. Mironov, A. S. Gusarenko, N. I. Yusupova. Stream handling large volume documents in situationally-oriented databases. In: *International Scientific Journal INDUSTRY 4.0. Scientific Technical Union of Mechanical Engineering "INDUSTRY 4.0"*. Vol. 3 (2018), Issue 5, pp. 240-244. ISSN 2534-8582.

[21] Cobo M. J., López-Herrera A.G., Herrera-Viedma E. A relational database model for science mapping analysis In: *Acta Polytechnica Hungarica*. 2015. V. 12. No. 6. P. 43–62.

[22] Arevalo C. и др. A metamodel to integrate business processes time perspective in BPMN 2.0 In: *Information and Software Technology*. 2016. V. 77. P. 17–33.

[23] Theodoro V., Abelló A., Thiele M., Lehner W. Frequent patterns in ETL workflows: An empirical approach. In: *Data & Knowledge Engineering*. Volume 112, November 2017, Pages 1–16. Crossref DOI link: https://doi.org/10.1016/j.datak.2017.08.004.

[24] Agh H., Ramsin R. A pattern-based model-driven approach for situational method engineering In: *Information and Software Technology*. 2016. V. 78. P. 95–120.

[25] Mironov V. V., Gusarenko A. S., Yusupova N. I. Situation-oriented databases: document management on the base of embedded dynamic model In: *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.

[26] Djukic V. etc. Model Execution: An Approach based on extending Domain-Specific Modeling with Action Reports In: *Computer Science and Information Systems.* 2013. V. 10. No. 4. P. 1585–1620.

[27] Shakhmametova G. R., Yusupova N. I., Mironov V. V., Zulkarneev R. Kh. Statistical and intelligent methods of medical data processing. In: I*nformation Technology in Industry*, Vol: 6, Issue 2, pp. 13-18, 2018.