IMPLEMENTING JSON OPERATIONS FOR IN-MEMORY DATA GRID AS PASS-THROUGH CACHE LAYER TO RDBMS

O. D. Borisenko and N. A. Lazarev
Ivannikov Institute forSystem Programing of the RAS, 109004, Russian Federation, Moscow, Solzhenitsyna st., 25
Lomonosov Moscow State University, 119991, Russian Federation, Moscow, Leninskie gory, 1

ABSTRACT
This article is a close look on the problem of storage and processing of high volume of JSON data. This format is in a strong demand therefore various DBMS provide support for this type as same as for XML. However, centralized data storing could be a “bottleneck” in application operation if high volume of data is processed. In this work we analyze capabilities of distributed In-Memory caching of disk database and provide handling of JSON-documents in Apache Ignite

Keywords: JSON-documents, XML and data grid

Cite this Article: O. D. Borisenko and N. A. Lazarev, Implementing JSON operations for In-memory Data Grid as pass-through cache layer to RDBMS, International Journal of Civil Engineering and Technology, 9(10), 2018, pp. 1033–1040.
http://www.iaeme.com/IJCIET/issues.asp?JType=IJCIET&VType=9&IType=10

1. INTRODUCTION
Industrial software involves storing and processing of high volume of data. For now, the most reliable way to store all the necessary information is the usage of relational data base which allows to built effective and expressive queries, to use data indexing and to control its consistency. However, with the development of software technology the demand for describing objects with substructure is growing. This demand may result in changes in software operation what essentially eliminate direct display into relational model as reorganization of data base in operation most often means closedown of application for a while. For describing complex objects such data formats as XML[1] and JSON[2] are used, where full-fledged support of XML specified in the SQL:2003[3] standard including special language of queries XQuery[4]. In the same time representing data in format JSON is more close-knit and easy-to-use, it also allows to handle documents on any platform without extra information about their structure therefore JSON-documents are extensively used in various
Implementing JSON operations for In-memory Data Grid as pass-through cache layer to RDBMS

web-application for the information exchange. The support of this data type and special functions for dealing with it are provided in some DBMS. As a result, the usage of semi structured data is possible with the retaining of the relational model features what in its turn increases opportunities of applications working with data bases and do not results in losing of advantages of such applications.

On the other hand, providing the increase of networking capacity and upgrading of computation nodes the centralization of relational DBMS and usage of on-disk storages in it result in appearance of a “bottleneck” in the organization of multiple user operation. A paradigm of distributed storages in RAM In-Memory Data Grid was proposed to solve this problem and usage of this technology for caching traditional databases allows to maintain control over data integrity.

For caching a relational DBMS two main approaches are used: “cache aside” and write-through caching. Unlike the first approach the write-through caching do not causes the application to interact with the main storage; the tasks of data update and requests processing are completed on the level of cache. It simplifies the application development, gives more capabilities for scaling and provides with data actuality in cache. In case of high volume of data insertion as with database initialization the write mode can be used directly in relational DBMS after what cache is updated in the background.

In this article existing instruments of distributed write-through caching of centralized relational DBMS are reviewed, and the support of the handling of JSON-documents directly in SQL-queries in the PostgreSQL syntax is provided.

2. REVIEW OF EXISTING INSTRUMENTS
In this part we review existing solutions in the field of data-handling in JSON-format in relational DBMS as well as possible methods of caching on-disk databases.

2.1. Support of JSON in relational databases
Processing of JSON-format isn’t described in SQL-standard, therefore to comparison of existing realization a description of required integration tools of XML-format contained in the standard SQL:2003 will be used. Let’s list main features from the point of view of the represented tasks:

1. SQL tables may contain in the columns JSON-type
2. Functions allowing developers generate JSON-documents directly in SQL-queries
3. Transformation of string data type in JSON and reverse operation
4. Support of CAST-operator both into JSON-type and out of it
5. Operations to check if the data is of JSON-type and to check correctness of the documents
6. Operators providing support for the processing of JSON-documents directly in SQL-queries
7. Support of stored-data indexing to allow quick access

Based on these features “Table 1” represents comparison of some DBMS providing support of JSON-type [5][6][7][8].
Therefore, most of relational DBMS keep the principles inserted when XML data type was implemented into SQL. Judging from given comparison MySQL and PostgreSQL projects stand out. PostgreSQL also supports JSON data binary storage format, provides wide range of facilities for JSON format documents handling directly in SQL-queries and at once it makes possible indexing of data being binary stored. It is a reason why this particular DBMS was chosen for further work.

### 2.2. In-Memory Caching

Remarkable price decline in the market of cached memory had the serious influence in the field of In-Memory technology, and plenty of various services providing this technology are affordable today, however, most often these projects addressing limited circle of tasks, but not the development of a universal database in the cached memory. In addition to it most of these projects support set of atomic data types designed for current task all without JSON support provided. Some projects include conversion of user defined data type into binary objects with the loss of all the type information.

Current task requires a service with open source code supporting data base caching with keeping relational model features, preserving date consistency when distributed be cluster and providing main ways of data access for applications. Beyond that another important aspects of proper choice are cluster in use extension facilities and support of various development platforms.

Let’s define criteria for comparison of instruments for In-Memory storage more precisely:

1. SQL support
2. Various SQL-dialects support
3. Caching operations supported
4. Data consistency with queries
5. Capabilities of indexing
6. System failure tolerance
7. Support of ACID transactions
8. Storage connection interfaces
9. Cluster extension facilities
10. Support of various development platforms
11. Facilities to store user defined data type
12. Open source code

On the basis of these criteria the comparison of some In-Memory databases providing in-memory caching [9][10][11][12][13] is displayed in the “Table 2”.

---

**Table 1** Comparison of capability of relational DBMS with regard to JSON

<table>
<thead>
<tr>
<th>Number of feature</th>
<th>Oracle</th>
<th>MySQL</th>
<th>MS SQL Server</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stored in strings</td>
<td>Yes</td>
<td>stored in strings</td>
<td>Yes, Binary Storage is possible</td>
</tr>
<tr>
<td>2</td>
<td>Incompletely</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Incompletely</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Incompletely</td>
<td>Yes</td>
<td>Incompletely</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>Yes</td>
<td>Yes</td>
<td>Incompletely</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>Full text search</td>
<td>No</td>
<td>No</td>
<td>For binary representation</td>
</tr>
</tbody>
</table>
Implementing JSON operations for In-memory Data Grid as pass-through cache layer to RDBMS

Table 2 Comparison of relational DBMS In-Memory caching tools

<table>
<thead>
<tr>
<th>Number of criterion</th>
<th>Oracle Coherence</th>
<th>Hazelcast</th>
<th>Terracotta</th>
<th>Pivotal GemFire</th>
<th>Apache Ignite</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SQL-similar language of queries</td>
<td>SQL-similar language of queries</td>
<td>SQL-similar language of queries</td>
<td>OQL</td>
<td>SQL:1999</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Additional adjustment is required</td>
<td>No</td>
<td>No</td>
<td>Most of dialects are supported</td>
</tr>
<tr>
<td>3</td>
<td>Read-through and Write-through</td>
<td>Read-through and Write-through</td>
<td>Read-through and Write-through, writes directly on the disk</td>
<td>Read-through and Write-through, writes directly on the disk</td>
<td>Read-through and Write-through, writes directly on the disk</td>
</tr>
<tr>
<td>4</td>
<td>Is guaranteed</td>
<td>Is not guaranteed</td>
<td>Is not guaranteed</td>
<td>Is not guaranteed</td>
<td>Is guaranteed</td>
</tr>
<tr>
<td>5</td>
<td>Indexing of single columns</td>
<td>Indexing of single columns</td>
<td>Indexing of single columns</td>
<td>Indexing of single columns</td>
<td>Full-fledged support of B-tree</td>
</tr>
<tr>
<td>6</td>
<td>Storing local data</td>
<td>Storing local data</td>
<td>No</td>
<td>Storing local data</td>
<td>Storing local data and indexes</td>
</tr>
<tr>
<td>7</td>
<td>Partial compatibility with ACID</td>
<td>ACID is not guaranteed in the distributed mode</td>
<td>ACID is not guaranteed</td>
<td>ACID is not guaranteed</td>
<td>ACID is not guaranteed</td>
</tr>
<tr>
<td>8</td>
<td>JPA</td>
<td>JCache interface</td>
<td>JCache interface</td>
<td>JCache, JDBC</td>
<td>JCache interface, JDBC, ODBC</td>
</tr>
<tr>
<td>9</td>
<td>Dynamic node adding</td>
<td>Dynamic node adding</td>
<td>Static cluster configuration</td>
<td>Dynamic node adding</td>
<td>Dynamic node adding</td>
</tr>
<tr>
<td>11</td>
<td>Binary Storage</td>
<td>Binary Storage</td>
<td>No</td>
<td>No</td>
<td>Binary Storage</td>
</tr>
<tr>
<td>12</td>
<td>Test version</td>
<td>Open source version with incomplete facilities range is available</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

On the basis of given comparison, the project Apache Ignite was chosen for integration of data handling in JSON-format.

This chapter describes the most important aspects of Apache Ignite working with write-through cashing DBMS under the current task.

3.1. Cluster initialization

Apache Ignite supports cluster’s scaling out in real-time mode, allowing usage of a static cluster configuration description as well as dynamic node search. It is also possible to set several virtual clusters on the same physical nodes. Computation nodes used in a cluster could be servers that are used for data storage and its internal processing and client that employ application to access data.

Even with the static setting, client nodes connections is made without reconfiguration and while the main cluster is working. For data storage, Apache Ignite uses the key/value model with cluster sharding via showing keys in multiply node identifiers. During write-through cashing of some
DBMS, database business case must be mapped as JAVA classes [14] in cluster configuration.

For further serialization, you need to indicate values upon which the cluster sharding is processed, describing the interaction with the main storage. On the ground of the data, necessary grids are created with the use of each cluster node, according to the following scheme:

- Active nodes are informed about adding a new one.
- Mapping SQL types to JAVA classes are processed and a structure of storage database in RAM is formed.
- Requests on creating required grids according to the formed database scheme are generated, system grids are also created.
- New grids are created.
- The rest cluster nodes are informed of availability.

Cache is possible after sets of cluster identifiers are generated to a certain state of configuration and the function of keys transformation of In-Memory storage to ids is complete. In this article we observed the work of In-Memory caching without replications because stability is achieved by disk caching.

3.2. SQL-requests processing

For processing SQL-requests, Apache Ignite was integrated with H2 Database Engine [15] tool implementing SQL:1999 [16]. In relation to the study of processing JSON-documents with write-through caching PostgreSQL an execution of SELECT DML-subset command (SQL) would be considered.

In this case we use Apache Ignite embedded interface in which original requests are translated into syntax H2 Database Engine, processed on client’s node, that initializes request’s execution and are divided into requests for local execution on nodes, containing required data and into one acquisition request. Resulting local record sets are transferred to a node, initializing execution of a request, on which received parts’ union is executed. That way when executing SQL-requests SQL dialect is implicitly used, it is implemented in H2 Database Engine, not in PostgreSQL. The other important feature of write-through caching mode is processing of data, that is located on cluster at the moment of request execution without cache data updating.

The reason for that limitation is the necessity to get included in records’ result keys, which is possible only with executed request in cached database. That would defeat the purpose of In-Memory storage.

4. JSON PROCESSING INTEGRATION

Adding a new data type and language constructions to Apache Ignite could be decomposed on the following steps:

- Implementation of required classes and methods and improvement of a request handler in H2 Database Engine;
- Implementation of JSON-documents storage
- Implementation of required classes and methods in Apache Ignite;
- Organization of an exchange of JSON-documents between Apache Ignite and H2 Database Engine.
Implementing JSON operations for In-memory Data Grid as pass-through cache layer to RDBMS

With the implementation of inner class H2 Database Engine, Java library fasternml.jackson, allowing serialization functions and processing JSON-documents is used. The following JSON-documents operators, PostgreSQL caching oriented, were added on the level of request handler:

- “->”, “->>”, “#>”, “#>>” – get a document’s attribute
- “@>”, “<@” – compare documents
- “?”, “?|”, “?&” – check document’s attribute availability.
- “||”, “-” – document’s modification

At the implementation of Apache Ignite classes and methods and organization an exchange between modules, problem of adding a new type of data in Apache Ignite and its mapping into data types of H2 Database Engine, an organization of serialization of input data and storage of processed serialized clusters were solved.

4.1. Testing

Running speed of implemented functions analysis based on pg_nosql_bechmark[17], developed for processing speed of JSON data in DBMS PostgreSQL analysis. Current version of the benchmark provides JSON-data generation function and execute requests, using access handler of document’s attributes. With the help of benchmark interface, a database, containing 40 million records, was generated. During testing, one of the proposed benchmark’s request, modified to minimize transferring of resulting data to client was used. Also requests, checking all implemented running functions for JSON-documents were used. Full list of processing requests in syntax PostgreSQL:

```
SELECT count(id) FROM table WHERE data->>'brand' = 'ACME';
SELECT count(id) FROM table WHERE data::JSONB ? 'name' AND data->>'name' = 'AC3 Case Red';
SELECT count(id) FROM table WHERE data::JSONB ?& array['type', 'name', 'price'];
SELECT count(id) FROM table WHERE data::JSONB ?& array['type', 'name', 'price'] AND data->>'type' = 'phone';
SELECT count(id) FROM table WHERE (data->>'limits'->>'voice'->>'n')::DECIMAL > 400;
SELECT count(id) FROM table WHERE (data#>>'{limits, voice, n}')::DECIMAL > 400;
SELECT count(id) FROM table WHERE data::JSONB?"color" AND data->>'color' = 'black' AND data::JSONB?"price" AND (data->>'price')::DECIMAL = 12.5;
SELECT count(id) FROM table WHERE data::JSONB@>'{"color":"black", "price":12.5}';
```

During testing each cluster node is a virtual machine controlled by hypervisor Xen with 92 Gb RAM, by three processor cores at the frequency of 2.1Hhz (Intel Xeon Gold 6152) and with provided physical server SSD. The results of request processing are compared with the results of request processing in PostgreSQL 10 received with optimized configuration. Time of request processing is shown at Pic.1 and table 3.
Table 3 Query execution time

<table>
<thead>
<tr>
<th>Startup</th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>q4</th>
<th>q5</th>
<th>q6</th>
<th>q7</th>
<th>q8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ignite, 4 nodes, sec.</td>
<td>488</td>
<td>980</td>
<td>501</td>
<td>613</td>
<td>497</td>
<td>496</td>
<td>776</td>
<td>508</td>
</tr>
<tr>
<td>Ignite, 8 nodes, sec.</td>
<td>261</td>
<td>518</td>
<td>265</td>
<td>330</td>
<td>265</td>
<td>268</td>
<td>413</td>
<td>268</td>
</tr>
<tr>
<td>Ignite, 16 nodes, sec.</td>
<td>134</td>
<td>261</td>
<td>134</td>
<td>167</td>
<td>135</td>
<td>134</td>
<td>206</td>
<td>133</td>
</tr>
<tr>
<td>PostgreSQL, sec.</td>
<td>240</td>
<td>245</td>
<td>281</td>
<td>285</td>
<td>250</td>
<td>248</td>
<td>301</td>
<td>268</td>
</tr>
</tbody>
</table>

During requests’ processing on every cluster’s configuration the same results as with requests’ processing in PostgreSQL were achieved.

5. ACKNOWLEDGEMENTS

This work is funded by the Minobrnauki Russia (grant id RFMEFI60417X0199, grant number 14.604.21.0199)

6. CONCLUSION

The article describes capabilities of write-through In-Memory cashing of traditional SQL-based engines, support of processing JSON-documents to Apache Ignite was implemented and testing of received tool based on pg_nosql_benchmark was made.

The testing shows that suggested implementation is better than PostgreSQL in terms of requests’ processing speed but it requires more resources to serialization of documents and organization of distributed data storage.

In our work, we implemented JSON-documents storage only in txt, which resources’ costs on serialization are much higher than storing in binary format. Therefore, current results can be improved on implementation of binary document processing. Besides, it will allow implementing JSON-documents indexation.

At the moment, only few of the main operators of JSON data have been implemented, to provide full support of PostgreSQL caching, a massive expansion of functional is required.
Implementing JSON operations for In-memory Data Grid as pass-through cache layer to RDBMS

REFERENCES


http://www.iaeme.com/IJCIET/index.asp 1040 editor@iaeme.com