Computer Science • 19(4) 2018 https://doi.org/10.7494/csci.2018.19.4.3088

Mateusz Piech
Wojciech Frącz
Wojciech Turek
Marek Kisiel-Dorohinicki
Jacek Dajda
Aleksander Byrski

MODEL FOR DYNAMIC AND HIERARCHICAL DATA REPOSITORY IN RELATIONAL DATABASE

Abstract *The aim of this research is to build an open schema model for a digital sources repository in a relational database. This required us to develop a few advanced techniques. One of them was to keep and maintain a hierarchical data structure pushed into the repository. A second was to create constraints on any hierarchical level that allows for the enforcement of data integrity and consistency. The created solution is mainly based on a JSON file as a native column type, which was designed for holding open schema documents. In this paper, we present a model for any repository that uses hierarchical dynamic data. Additionally, we include a structure for normalizing the input and description for the data in order to keep all of the model assumptions. We compared our solution with a well-known open schema model – Entity-Attribute-Value – in the scope of saving data and querying about relationships and contents from the structure. The results show that we achieved improvements in both the performance and disk space usage, as we extended our model with a few new features that the previous model does not include. The techniques developed in this research can be applied in every domain where hierarchical dynamic data is required, as demonstrated by the digital book repository that we have presented.*

Keywords JSON, relational databases, EAV, CQRS, PostgreSQL, open schema model

Citation Computer Science 19(4) 2018: 479–500

1. Introduction

When it comes to choosing the best-suited database engine for dynamic data with a hierarchical structure (e.g., the data in the book repository described in Section 3), the answer is obvious at first glance – using NoSQL databases. This category includes a document database, for example, which is a natural approach for storing free schema documents based on the fact that they are well-supported [12]. However, in some cases (e.g., a digital resource repository), a document database cannot be used. Document database limitations include (but are not limited to) a lack of transactions, a lack of data schema control, and poor support of the relationships between the data. All of these features are provided by relational databases.

There are several data models in relational databases that allow for the storage of open schema (also known as a generic data model [6]). This solution allows us to modify the data model without changing the relational database schema. A couple of the best-known solutions are Entity-Attribute-Value (EAV) [11] and inverted index [18]. Unfortunately, these lack schema readability; moreover, CRUD operations are not as fast here as they are in the traditional model. However, many relational databases recently started to support JavaScript Object Notification as a native type. JavaScript Object Notification (JSON) [2] is a common data-interchange format designed primarily to serialize and transmit data over a network. It has many advantages: it is easy for humans to read and write as a text format, relatively lightweight (when compared to other alternatives), and supported in many programming languages. It has an impact on storing dynamic data. According to a "new approach to storing dynamic data in relational databases using JSON" [13], the performance improvement was achieved when compared to the EAV model. The solution was also compared with MongoDB (a representative document database), and the obtained results were comparable. Although storing data in JSON has many advantages, it is not free of imperfections. Representing dynamic objects' schemas is difficult, and the presented solution increases the complexity of query creation.

The main goal of this research is to prepare an open schema model that can be used for creating a digital source repository. This domain uses data that possesses a hierarchical structure. Additionally, for every type of data repository, it is required to know the objects' structure to create a query language, so the model should have facilities to manage dynamic schema in a transparent way. Moreover, the repository must keep cohesion between the objects and its references, so it should be able to handle constraints on any hierarchical level, thus maintaining data consistency. A partial solution is provided by the JSON features – the rest of the solution requires preparation and testing. Developing an open schema model with a hierarchical structure should be possible to adopt in every field of science and technology (where such functionality is needed); so, the developed model must be universal.

This paper presents our model for a repository with hierarchical data in a relational database. It comes along with a description of the terms for the repository elements. These terms were used to create a component for keeping and maintaining

the objects' structure. It was designed to create constraints on the relationships on any level in the hierarchy, which is a feature that is missing in the existing models. We included a universal database schema for the model, which can be applied to any repository along with a prepared object structure. We also described an architecture for the final solution that was used in our benchmark. The product of the research was tested and compared with the entity-attribute-model data model to measure the achieved improvement in the time and disk space usage. The results show that our model can be applied to any field where dynamic objects with a hierarchical representation are needed.

Summarizing, the research goals are as follows:

1. Create a dynamic model for hierarchical data with support of JSON in which the relationships and dynamic schema are maintained by a database engine.
2. Enable the creation of a DSL for querying data by keeping the structure of the dynamic data.
3. Improve the query performance and disk space usage when compared to the existing entity-attribute-model.

In the next section, we describe the current state of the art in the field of adapting JSON into relational databases. We also include an overview of the existing open schema model solutions. Then, we move on to presenting the terminology needed for the generic data model for a repository along with its description. Later, we describe the application of the created model to the repository. Finally, we present an evaluation of the model in which we compared our model with the entity-attribute-value model. At the end of the paper, we include our conclusions and the next steps of our research.

2. Related work

The question of how to store data with a dynamic structure in a relational database was asked a long time ago. It appeared when the diversity of the collected data caused problems with their storage in a fixed database schema. One of the answers for preparing an open schema storage approach was the development of the entity--attribute-value [11] and inverted index [18] models, which were applied to many fields of science. However, these solutions were not perfect, so there were many attempts to improve upon them. The experiments were made on clinical databases [3] and bio-medical databases [11], for example. These ideas led to slight performance improvements; however, as mentioned earlier, the problems with the readability of the schema that resulted in the difficulty of creating queries were not resolved. The performance was still far from perfect, so relational database engines started to introduce semi-structured objects as a native type. In the first approach, they introduced the ability to store XML as a native type (Microsoft SQL Server 2005) [14], but its heavy format increased disk usage [17].

Most of the relational databases have already introduced the ability to store JSON data:

- PostgreSQL 9.2 (2012),
- Oracle Database 12c Release 1 (2013),
- MySQL 5.7 (2015),
- Microsoft SQL Server 13.00 (2016).

This decreases the functional and performance gap between SQL and NoSQL [9].

Discussions on the utilization of JSON as a type for storing data with a dynamic structure were conducted before database engines introduced it as a native type. The most successful attempts to provide this feature consisted of custom implementations of storing the JSON files. The most promising results were presented in "Enabling JSON Document Stores in Relational Systems" [1] and "Sinew: a SQL system for multi-structured data" [16], where two systems were introduced. The papers claim that relational databases provide better performance than document databases in some cases. They highly outperform the entity-attribute-value approach in every test. Additionally, the extension of these benchmarks in "JSON data management: supporting schema-less development in RDBMS" [8] has shown that introducing indexes for JSON documents improves performance.

New perspectives in relational database engines opened up new opportunities for a creating a repository for dynamic data in relational engines. The need for the creation of such a system was diagnosed long ago. One of the well-known solutions was a system called Lore, which was introduced in "Lore: A database management system for semi-structured data" [10]. In this research, all of the aspects of a database management system (including storage management, indexing, query processing and optimization, and user interfaces) were revised. As a result, the created system was designed to store any data structure; however, it required several adjustments outside the database engine to allow it. Nowadays, the introduction of JSON as a native type can simplify these settings and can be moved into a database engine, which should improve the performance.

Designing such a repository will affect many fields of science. One of these is criminal data, where the number of possible objects and relationships between them is enormous. The adaptation of JSON was successfully presented in the "new approach for storing dynamic data in relational databases using JSON" [13], where the tests of the presented model showed a significant advantage over document databases in real-use cases.

3. Modeling dynamic schema for generic repository

The main goal of this research is to create a model for the digital source repository as a library of books and research papers. However, we always bear in mind that we want to achieve wider applicability. In the beginning, we examined this problem;

as a result, we observed that the initial use-case scenario can be broken down into several elements. We divided them into two categories:

- **resources** – which are physical objects,
- **relationships** – which are connections between objects.

During the evaluation, we focused on two pairs of resources and relationships; i.e., (*Book*, *Author*), and (*See also*, *Related*). These elements can have an enormous variety of attributes, including multiple values of the same attribute (e.g., titles). The simple example of a schema for *Book* and *Author* and the relationship between them is shown in Figure 1. Undoubtedly, the presented case is simplified, but it is adequate for understanding and addressing the problem.

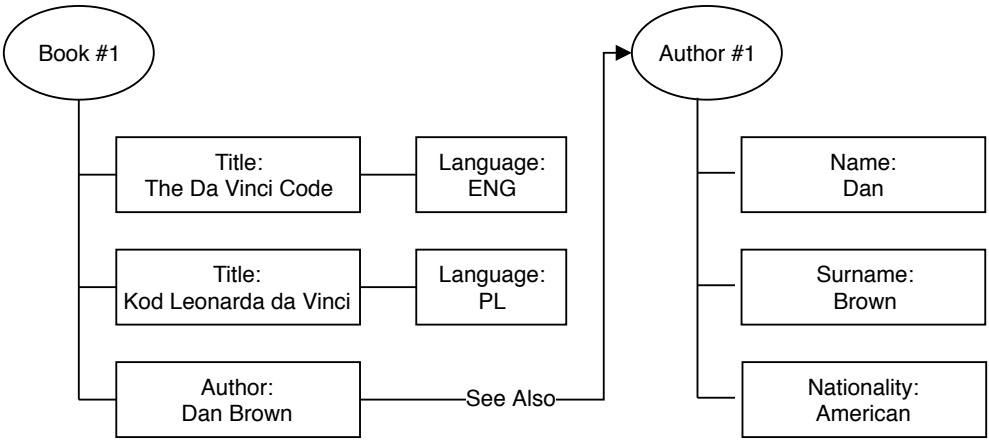

Figure 1. Graphical schema of two dynamic objects and relationships between them for digital source repository

3.1. Open schema model before JSON

For the verification of our research results, we selected an open schema model that, with a few adjustments, could be used for implementing a repository with a dynamic and hierarchical structure. Therefore, we applied our requirements to the EAV data model. This forced an extension of the default model by adding two extra tables: one for dictionary optimization, and one for resource linking.

Our goals require two types of relationships:

1. **Resource-Resource** – simple relationship between two resources; e.g., *Book#1 has a relationship with Book#2.*
2. **Attribute-Resource** – relationship between any attribute and external resource; e.g., *Attribute Author in Resource Book#1 refers to Author#1.*

In the schema in Figure 2, these types of relationships are realized by using the `EntityRelation` table. The resource-resource relationship is represented by a row

with `valueId` equal to the `NULL` value. The attribute-resource relationship row has the `valueId` set to the identifier of the relationship attribute.

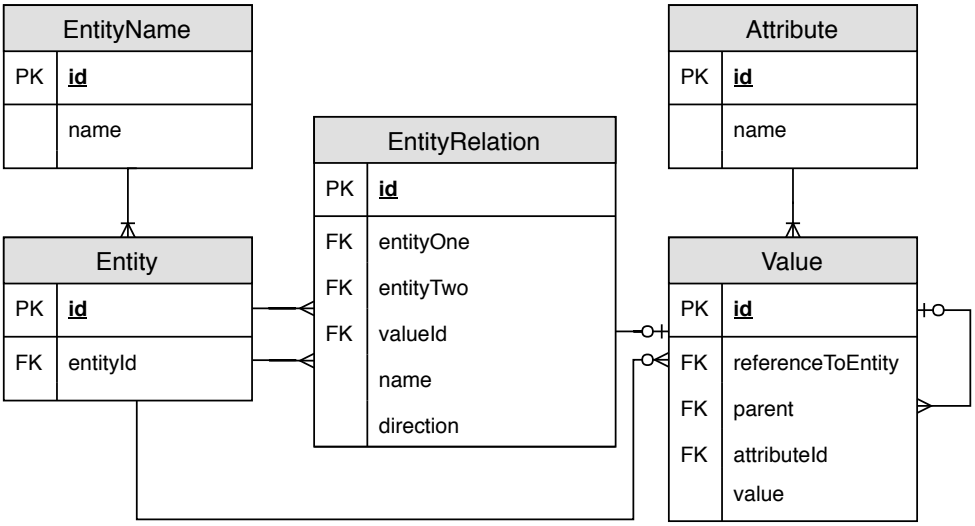

Figure 2. Implementation of dynamic hierarchical model in EAV

The hierarchical structure of the attributes is implemented in two steps. The reference of the first-level attributes to the resource is realized by setting `Value#referenceToEntity` to the resource ID from `Entity#id`. The tree structure of the attributes is created by setting `Value#parent` to the parent attribute. This also means that `NULL` indicates the first level.

Summarizing, we have designed the five following tables in the EAV model:

- `EntityName` – dictionary table that contains names of resources.
- `Entity` – table that holds information about resource id and its type.
- `Attribute` – dictionary table that contains names of attributes.
- `Value` – table that holds string value of an attribute, relationship in hierarchy (`parent`), resource owning value (`referenceToEntity`), and attribute id.
- `EntityRelation` – all relationships between resources, with the information of `name`, `direction`, and type of relationship. If `valueId` is set, then type of relationship is attribute-resource; otherwise, resource-resource.

It is worth noticing that this schema facilitates the possibility of observing the structure of the resource. However, it does not enforce consistency in the structure between two resources of the same type. This is caused by not utilizing any control schema mechanism. This implicates the problem of determining which attributes appear in each resource type. Thus, the creation of an efficient DSL for queries is almost impossible due to the huge number of joins required to examine the entire structure.

3.2. Open schema model after JSON

An analysis of the existing solutions resulted in the discovery of weaknesses in the previous model and gave us the inspiration for resolving them. We began the process of developing the new model by defining the terminology required for the dynamic data repository. As a result, we introduced **Metadata**, **Template**, **Resource**, and **Relationship** concepts. They can be divided into two categories: data storage and configuration. However, Relationship is designed both for holding data content and ensuring the repository integrity.

The primary unit of the repository is Metadata. The main idea of Metadata is that it is arranged into a tree [15] with an unlimited number of children for the parent. This allows us to represent data with a hierarchical structure. Metadata is used in two meanings:

- Definition – which is used to control and maintain the schema. It contains information about itself: label, information about position in a hierarchical tree (parent, ordinal number), type control (i.e., type of Metadata value – e.g., date, text, number), constraints (e.g., regular expression, number range, or date format), and also a description and placeholder (if needed).
- Value – as an element of the content. It is defined as a key-value pair where the key is a path in the hierarchical tree and the value stores its data.

The next piece of the repository configuration is Template, which connects the metadata to each other into a logical unit. Template is defined by a label and a set of metadata roots. It behaves like a root in a tree. It is worth noticing that the term "Template" refers not only to the aforementioned first level of attributes but also the entire sub-tree of the nodes reachable from such roots. It is used to create an identity for an input; e.g., *Book*, *Author*, and *See Also*, which is demonstrated in Figure 1.

The major element in the repository (Resource) is responsible for holding the normalized and validated input pushed into the repository. The validation mechanism checks whether the content matches the template specified by the attached label. It is not required to fill all of the metadata defined by Template, but a valid content must not contain any metadata not included in Template. The normalization process is responsible for transforming the data into a unified structure. This facilitates storage and analysis of the input data. The designed structure is presented in Figure 3. It starts with a root named `content` and refers to the first level of Metadata by labels. They are combined by a list of input values that can refer to other resources and can hold the next levels of metadata according to the hierarchical structure. The next levels follow the same rules. The example of normalized input from Figure 1 is presented in Figure 4.

The last element of the repository is named Relationship. This element is responsible for controlling the schema and holding the data. This piece of the repository is similar to Resource because it holds normalized content and has its own template. The configuration part of the repository in this element is based on references to resources in the relationship and direction. It might also reference Metadata.

Relationships are created directly and indirectly: directly, when a relationship is of the *Resource-Resource* type and is pushed into the repository; and indirectly, when a relationship is of the *Attribute-Resource* type and is inferred from Metadata. The second type ensures data integrity on any level in Resource's hierarchy.

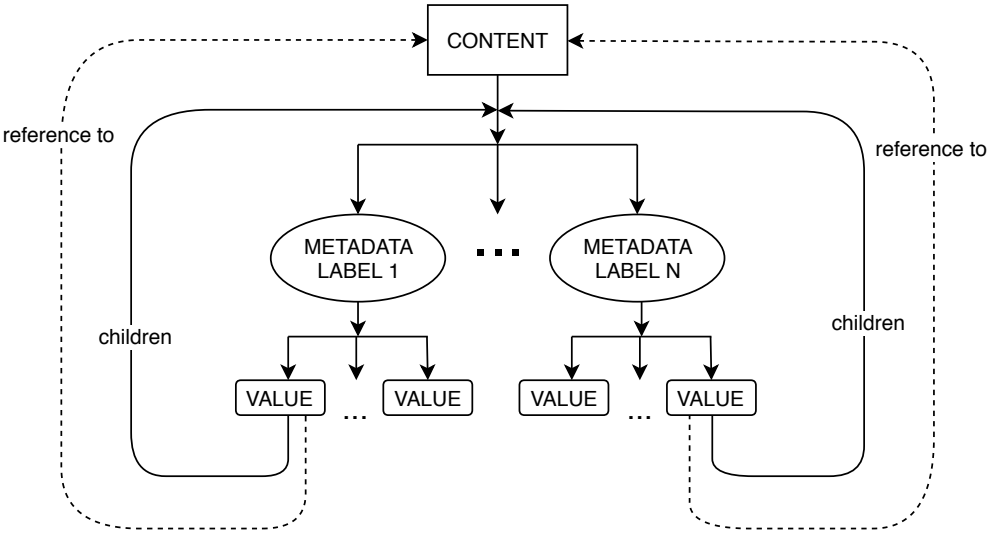

Figure 3. Visualization of structure used for normalization and validation of repository input data

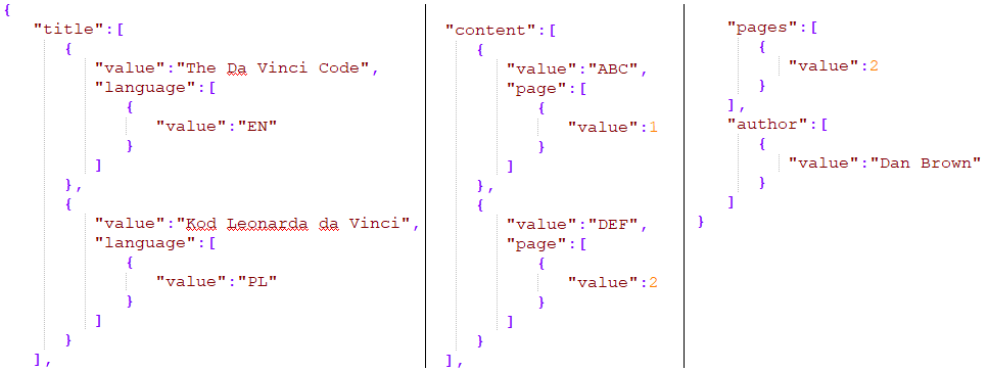

Figure 4. Digital repository input normalized to valid structure

A relational database schema that enables us to store the designed model is shown in Figure 5. It consists of four main tables. Each element possesses its own table; however, this implementation of the schema does not satisfy all of the requirements posed at the beginning of our research.

To achieve the remaining goals, we propose an extra layer with the mechanism of validating and transforming the data pushed into the repository.

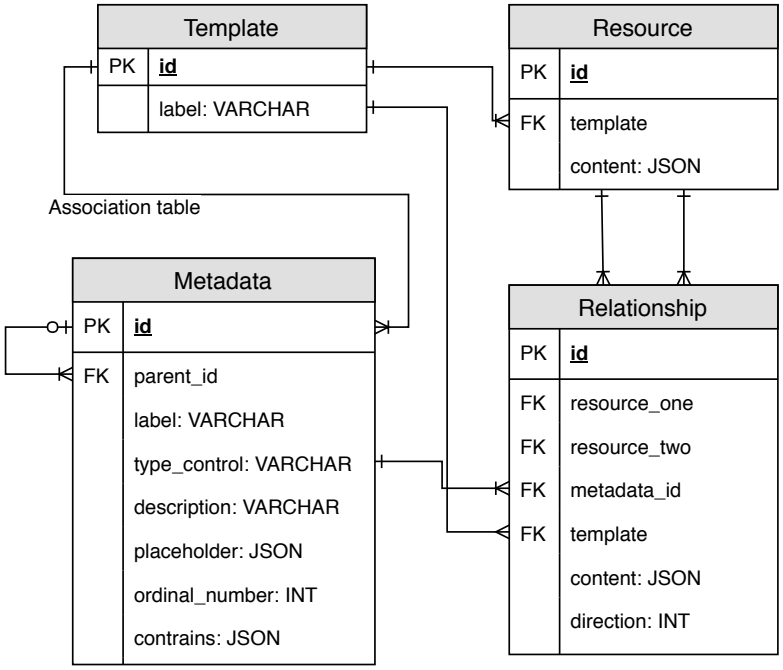

Figure 5. Model schema for dynamic repository in relational database

The algorithm consists of the following steps:

- before inserting the document, ensure its template already exists;
- push Data into Repository;
- validate and transform content:
  - validate Metadata:
    - ensure content has proper structure as described by Template,
    - ensure values have proper type;
  - transform content:
    - JSON document to content structure,
    - determine all indirect relationships of attribute-resource type and create them;
- save all created entities in database.

The steps are applied for both Resource and Relationship. The presented data flow can be implemented in the database engine as a procedure or trigger (or even outside the database if needed).

However, there is one main rule that needs to be respected: the whole process must be carried out as a single transaction to preserve data consistency. Last but not least, the unmentioned case of deleting Resource is subject to the constraints of the schema. So, there is no possibility to delete a resource that is being referred to – if there is a relationship between resources on any level (attribute/resource-resource), the constraints in the relationship table will be checked, and the issue will need to be resolved by the repository user.

4. Model application in repository

In the previous section, we presented a design of the model for a hierarchical dynamic repository. However, it is not usable as a stand-alone solution; it must be utilized by other components. Therefore, we prepared an entire architecture for our digital resource repository (as presented in Figure 6). The ideas, however, are general and can be applied to any domain.

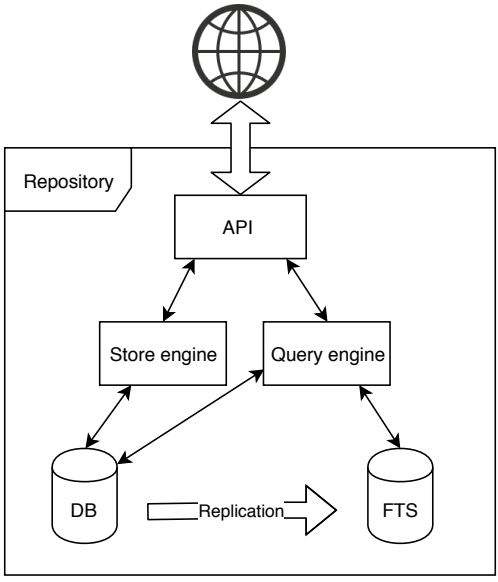

Figure 6. Final repository architecture

As it can be observed, we divided it into several components. The first one is Interface over the Internet (API) with the necessary methods for handling the data processing. The two remaining services are Store Engine and Query Engine. Store Engine is responsible for validating, transforming, and inserting data into the correct tables. It is also responsible for creating and keeping the relationships between the data (already mentioned in the data flow steps). Query Engine is responsible for selecting the optimal query plan for executing the search action.

To accomplish faster searching with the possibility of using a full-text search, we decided to introduce a pattern: command query responsibility segregation. According to [5], CQRS is a design pattern with a clear separation of reading and modifying the data. Not only did we decide to utilize this technique by using two different engines, but we also separated the query types between them. In the evaluated resource repository, we have two search types:

1. search for content – e.g., search for book that has word 'digital' in any lexical form in Metadata value,
2. search for relationships – e.g., find all books that are connected by relationship *See also*.

Thus, we ended up with two database engines: one with a built-in function for a full-text search, and the second one – a relational database with native JSON support in which we developed the model that keeps all of the data. It is also worth mentioning that populating the FTS engine using the data kept in the relational engine is done by a replication task.

5. Evaluation of model

In order to achieve our research goals, the next step was to compare the EAV model and the one we designed in the previous section. We started with selecting the tools and database engines. Then, we prepared data sets and test use cases. Finally, we analyzed the results of the executed tests.

5.1. Tools and database engines

Before we proceeded to the benchmark tests, we had to select the relational database engine to compare our solution with the previous one. We decided to choose PostgreSQL, which had promising results in many research projects [1, 13, 16]. Both models were developed with the same engine (Version 10.4 – the latest at the time of the research). Both implementations were also tested in the same run configuration to obtain the most adequate comparison. The machine employed in the tests had a 3.2 Ghz quad-core Intel i5-4460 processor with 16 GB of memory and 256 GB of solid-state storage. During our test, we focused on the evaluation of the saving process and queries about the relationship and accessing content. We decided to omit the content query performance comparison since the data in both models can be replicated to the search engine – e.g., Elasticsearch or Solr [7].

As a full-text search engine, we chose Elasticsearch 6.2. The major reason for selecting this document database was its popularity in DB-engine rankings [4] and the supporting tool – Logstash, which helped us create replication between the relational database and the full-text search engine. Moreover, it has a feature crucial for our model – it allows us to create a dynamic mapping for resource content. This is relevant because we wanted to automate the indexing process by auto-applying analyzers for different data types on all hierarchy levels without modifying the index configuration during runtime.

5.2. Test preparation

The process of preparing the data sets for the tests was based on goals – we wanted to observe how increasing the number of resources while keeping the number of relationships constant (and conversely) affected both models. For this purpose, we created a data generator; it was based on a random selection of attributes from the complete resource structure and filled by randomly selected values of data (e.g., names, last names, lengths).

Thus, we generated 12 sets of data – 6 sets with different numbers of resources (0.5 M, 1 M, 1.5 M, 2 M, 2.5 M, 3 M) and 1 million relationships, and 6 sets with different numbers of relationships (0.5 M, 1 M, 1.5 M, 2 M, 2.5 M, 3 M) and 1 million resources. Each data set was generated in two variants that differed by the maximum depth of each hierarchy tree: 2 (H2) and 5 (H5). In each data set, we created two types of resources (*Book* and *Author*) and two types of relationships (*See also* and *Related*).

For the benchmark, we selected three queries that were supposed to compare the models at various levels.

(Q1) In the first query, we wanted to test the difference for the request for the values of the attributes of the resources related to a specified resource. Therefore, we implemented the following use case: for the given books, find the books titles that have relationships with them.

(Q2) In the second query, we wanted to test the difference in the requests for the resources related to another resource's metadata. So, we implemented the following use case: find the books of a given author.

(Q3) As the last-use case, we wanted to obtain data from Resource, so we created the following query: obtain the first pages of 100 selected books.

For each data set, we executed each query ten times and computed the median. The implementation of the selected queries in both models is included below.

```
-- Implementation of Q1 in EAV
SELECT value.value FROM entityRelation
JOIN value ON value.referenceToEntity = entityRelation.entityTwo
WHERE value.attributeId = ? AND entityRelation.entityOne = ?
AND entityRelation.name = ?

-- Implementation of Q1 in developed model
SELECT resource.content #> '{title, 0, value}' FROM resource
JOIN relationship ON relationship.resource_two = resource.id
WHERE relationship.resource_one = ? AND relationship.template = ?;

-- Implementation of Q2 in EAV
SELECT entityRelation.entityOne FROM entityRelation
JOIN value ON entityRelation.valueId = value.id
WHERE value.attributeId = ? AND entityRelation.entityTwo = ?
```

```
-- Implementation of Q2 in developed model
SELECT relationship.resource_one FROM relationship
WHERE relationship.metadata_id = ? AND relationship.resource_two = ?

-- Implementation of Q3 in EAV
SELECT v.value FROM value v
JOIN value v2 ON v.id = v2.parent
WHERE v2.attributeId = ? AND v.attributeId = ?
AND v2.referenceToEntity IN (?);

-- Implementation of Q3 in developed model
SELECT content #> '{page_content, 0, value}' FROM resource
WHERE template_id = ? AND id IN (?);
```

5.3. Test results

In the beginning, we measured the time it took to insert the data for both models. In the case of our model, the measurements included all of the steps: validation, transforming the data to a valid structure, and creating all of the needed indirect constraints for the relationships. Additionally, we measured the size of the database on the disk for each data set. In Figure 7, we presented the results for the data sets with a constant number of relationships (Figure 7a) and with a constant number of resources (Figure 7b). In both cases, we calculated how many times we achieved better results in our model as compared to the EAV model.

An analysis of the results from Figure 7 led us to several conclusions. Increasing the number of resources with constant numbers of relationships does indeed improve the insertion time when compared to the EAV model. Increasing the max hierarchy level also helped us obtain better results. However, the constant number of relationships brings us different results. In this case, we see a slight decrease in the relative difference between the models – since the relationships do not have any attributes, the benefits of our solution become negligible. However, our model is capable of holding the attributes of the relationships, whereas the EAV model is not. It is worth noticing that the disk space usage is correlated with the insertion time. The speedup is correlated with the weight of the JSON documents – which is noticeable in a comparison of the time results and disk space in H5 and for constant numbers of relationships for 1 M, 1.5 M, and 2 M.

In the next step, we ran three prepared queries on each data set. These results are presented in Figures 8 and 9. In Q1, we obtained a speedup compared to the EAV model. The efficiency gains are partially due to accessing the JSON files in our model instead of the `Value` table in the EAV model. The most important factor, however, is the better modeling of the relationship types (Template compared with field `EntityRelation#type`). This is particularly visible when the number of relationships increases (Figures 8b and 9b).

In Q2, we checked a simpler version of Q1 – it does not require access to a resource's content. As a result, the improvement measured in this query is not as good as in Q1, but an improvement is still achieved. This was expected, since the implementation of the query is similar in both models. The primary advantage of our solution is its lower disk space usage.

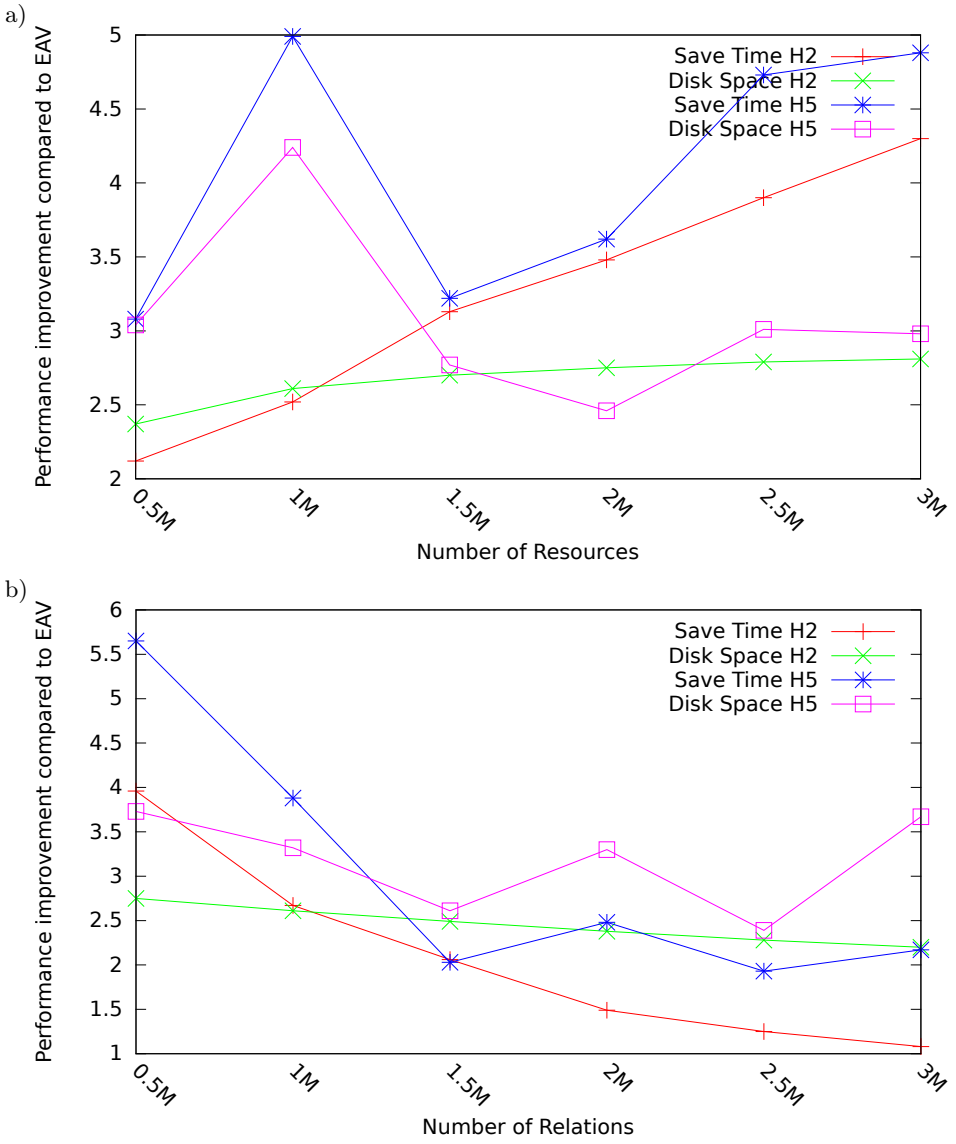

Figure 7. Measurements obtained during insertion process compared to EAV model: a) performance improvement for operations on data with constant number of relationships; b) performance improvement for operations on data with constant number of resources

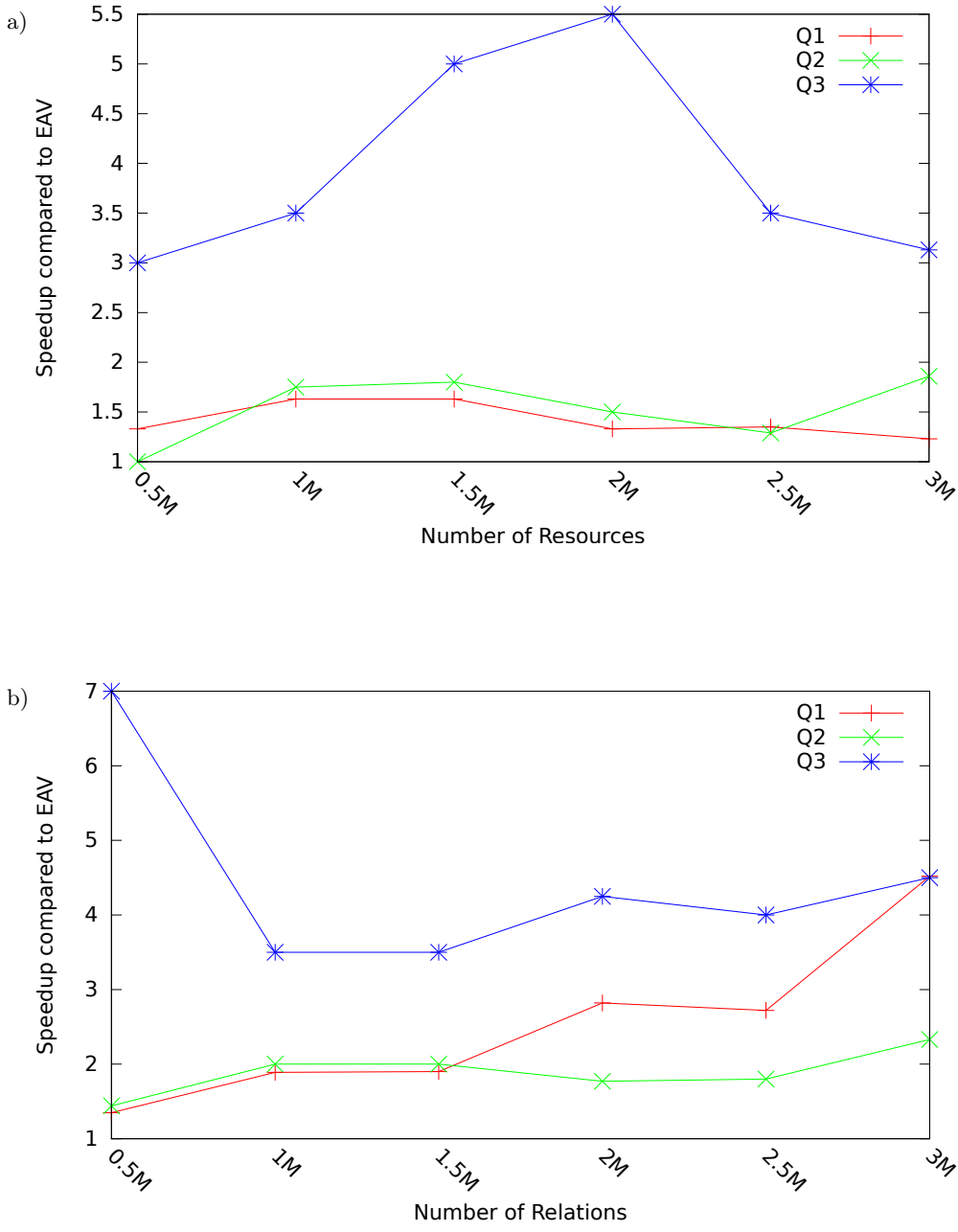

Figure 8. Measurements obtained during running queries and compared with EAV model: a) median of speedup for queries(H2) with constant number of relationships; b) median of speedup for queries(H2) with constant number of resources

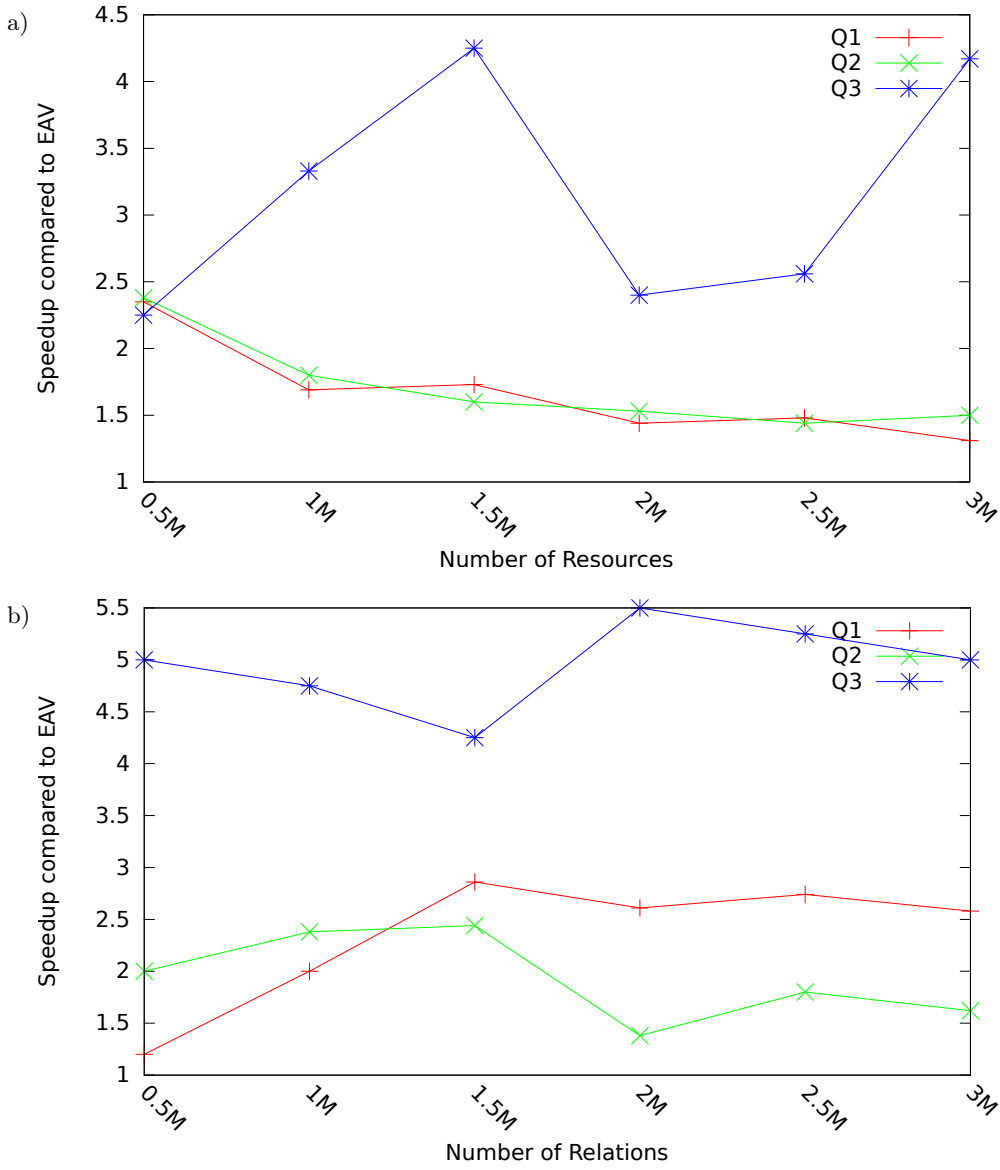

Figure 9. Measurements obtained during running queries and compared with EAV model: a) median of speedup for queries(H5) with constant number of relationships; b) median of speedup for queries(H5) with constant number of resources

In Q3, we tried to examine the case of extracting a value from the structure. In this query, we observed the highest (stable) speedup when compared to the EAV model. In this case, it is clear that increasing the max level of a hierarchy provides

better results. This is due to the use of JSON. The results of these measurements and all of the other tests are presented in Tables 1, 2, 3, and 4.

Table 1
Median of measurements for operations for data sets with constant numbers of relationships

	Our model		EAV	
Results	Insert Time	Disk Space	Insert Time	Disk Space
H2				
0.5 M	00:08:26	0.72 GB	00:17:53	1.71 GB
1 M	00:13:08	1.25 GB	00:33:09	3.27 GB
1.5 M	00:15:50	1.78 GB	00:49:37	4.82 GB
2 M	00:18:35	2.32 GB	01:04:45	6.38 GB
2.5 M	00:20:36	2.85 GB	01:20:21	7.93 GB
3 M	00:22:13	3.37 GB	01:35:37	9.49 GB
H5				
0.5 M	00:09:25	0.95 GB	00:29:02	2.88 GB
1 M	00:17:01	2.02 GB	01:24:56	8.56 GB
1.5 M	00:15:40	1.82 GB	00:50:23	5.06 GB
2 M	00:17:37	2.55 GB	01:03:52	6.28 GB
1.5 M	00:21:47	3.49 GB	01:42:58	10.49 GB
3 M	00:22:56	3.70 GB	01:51:50	11.04 GB

Table 2
Median of measurements for operations for data sets with constant numbers of resources

	Our model		EAV	
Results	Insert Time	Disk Space	Insert Time	Disk Space
H2				
0.5 M	00:08:07	1.16 GB	00:32:10	3.19 GB
1 M	00:12:21	1.25 GB	00:32:55	3.27 GB
1.5 M	00:16:54	1.34 GB	00:34:47	3.34 GB
2 M	00:23:20	1.44 GB	00:34:49	3.42 GB
2.5 M	00:28:56	1.54 GB	00:36:10	3.50 GB
3 M	00:37:19	1.62 GB	00:40:24	3.57 GB
H5				
0.5 M	00:10:25	1.61 GB	00:58:53	6.02 GB
1 M	00:15:06	1.74 GB	00:58:37	5.76 GB
1.5 M	00:18:53	1.42 GB	00:38:19	3.71 GB
2 M	00:26:37	1.94 GB	01:05:56	6.41 GB
2.5 M	00:29:49	1.62 GB	00:57:34	3.87 GB
3 M	00:40:11	2.33 GB	01:27:15	8.54 GB

Table 3
Median of measurements for execution on queries for data sets with constant numbers of relationships

Results	Constant numbers of relationships											
	H2						H5					
	Our model			EAV			Our model			EAV		
	Q1 [ms]	Q2 [ms]	Q3 [ms]	Q1 [ms]	Q2 [ms]	Q3 [ms]	Q1 [ms]	Q2 [ms]	Q3 [ms]	Q1 [ms]	Q2 [ms]	Q3 [ms]
0.5M	9	4	1	12	4	3	9	4	2	20	10	5
1M	8	4	2	13	7	7	13	5	3	22	9	10
1.5M	8	5	2	13	9	10	11	5	2	19	8	9
2M	11	6	2	14	9	11	13	9	5	18	13	12
2.5M	12	7	3	16	9	11	12	9	5	17	13	12
3M	13	7	4	16	13	13	13	7	3	17	11	13

Table 4
Median of measurements for execution on queries for data sets with constant numbers of resources

Results	Constant numbers of relationships											
	H2						H5					
	Our model			EAV			Our model			EAV		
	Q1 [ms]	Q2 [ms]	Q3 [ms]	Q1 [ms]	Q2 [ms]	Q3 [ms]	Q1 [ms]	Q2 [ms]	Q3 [ms]	Q1 [ms]	Q2 [ms]	Q3 [ms]
0.5M	9	5	1	12	7	7	10	4	2	12	8	10
1M	10	5	2	18	9	7	11	4	2	21	10	10
1.5M	11	4	2	20	8	7	11	5	2	32	11	9
2M	11	7	2	31	12	9	16	7	2	41	9	11
2.5M	13	5	2	34	9	8	14	5	2	37	9	11
3M	11	5	2	48	11	9	18	7	2	47	11	10

6. Conclusion and future work

At the beginning of the paper, we highlighted that we wanted to prepare an open schema model that can be used in creating a digital source repository. Not only did we achieve applicability to the digital repository, we also developed a general solution that can be used in all fields of science and technology where hierarchical dynamic data is utilized. Moreover, we managed to produce a model with a few extra features

that were not included in our previous open schema solution. Summarizing, our solution supports the following:

- **open schema model for hierarchical data** – by using JSON as data type for dynamic objects,
- **information about resource structure** – by introducing schema configuration tables (Metadata and Template) to keep all information about resource structure with validation component for data integrity,
- **consistency of data relationships** – by introducing relationship table and all needed constraints,
- **possibility of creating relationship on any level** – by creating necessary fields in relationship table to maintain two types of relationships: resource-resource and attribute-resource.

After the design process, we proceeded to test our model. For this purpose, we implemented an entity-attribute-value and compared it with our solution in several different usage scenarios. We evaluated the process of inserting data and three queries that were supposed to demonstrate how the performance of a relationship search and extraction of data differ between the two models. The prepared data sets were combined with two categories of data: a varying number of resources with a constant number of relationships and a varying number of relationships with a constant number of resources. The results led us to some valuable conclusions. We observed that, by using JSON for storing dynamic data, we decreased the used disk space, which resulted in decreasing the times of inserting the data and executing the queries. The advantage of our solution when compared to EAV grows with increasing both the number of resources and the max hierarchy level. Due to the simplification of the schema, we also observed a speedup in the tests with a constant number of resources and varying number of relationships despite the fact that our model and EAV are relatively similar in this case – we only added attributes that were not used. The results prove that we can apply this model to a real application without concerns about its performance.

Although our model is universal and complete for the presented purpose, we have a few ideas on how we can improve this repository. First of all, we are going to introduce the possibility of executing graph queries for the relationships, seeing as the performance of the existing algorithms leaves much to be desired. We believe that creating an algorithm based on a dynamic JSON structure can provide a significant performance boost. We hope that such an algorithm will be comparable in performance to the graph databases under some circumstances. The second idea is based on CQRS, which was introduced in the section on repository architecture for separating queries type. We have yet to develop a mechanism that allows us to execute queries involving both relationships between the resources and their contents. We are considering two strategies: first-query data (Elasticsearch) or first-query relationships (PostgreSQL). We do not know which approach should be used for a mixed query yet – e.g., find authors for books with a title containing 'Foo'. We expect both ideas to

be viable; implementing either should significantly increase the applicability of our solution to domains that require complex queries.

Acknowledgements

The research presented in this paper is supported by the R&D project under the auspices of EU Funds and the Polish Ministry of Digitization: European technological legacy – dissemination of historical and contemporary technical science publications in an innovative IT system, AGH University of Science and Technology, 2016–2019.

References

[1] Chasseur C., Li Y., Patel J.M.: Enabling JSON Document Stores in Relational Systems. In: *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, NY, USA, June 23*, pp. 1–6, 2013.

[2] Chen H.: Javascript object notation schema definition language. US13596694, 2014. https://www.google.com/patents/US20140067866.

[3] Chen R.S., Nadkarni P., Marenco L., Levin F., Erdos J., Miller P.L.: Exploring Performance Issues for a Clinical Database Organized Using an Entity-Attribute--Value Representation, *Journal of the American Medical Informatics Association*, vol. 7(5), pp. 475–487, 2000. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC79043/.

[4] DB-Engines Ranking of Search Engines. https://db-engines.com/en/ranking/search+engine.

[5] Fowler M.: *CQRS*. Martin Fowler's Blog, 2011. https://martinfowler.com/bliki/CQRS.html.

[6] Johnson S.B.: Generic data modeling for clinical repositories *Journal of the American Medical Informatics Association*, vol. 3(5), pp. 328–339, 1996. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC116317/.

[7] Kılıç U., Aksakalli I.K.: *Comparison of Solr and Elasticsearch Among Popular Full Text Search Engines and Their Security Analysis*. https://doi.org/10.13140/RG.2.2.24563.32803.

[8] Liu Z.H., Hammerschmidt B., McMahon D.: JSON Data Management: Supporting Schema-Less Development in RDBMS. In: *SIGMOD'14 Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 1247–1258, 2014. https://doi.org/10.1145/2588555.2595628.

[9] Liu Z.H., Hammerschmidt B., McMahon D., Liu Y., Chang H.J.: Closing the functional and Performance Gap between SQL and NoSQL. In: *SIGMOD'16 Proceedings of the 2016 International Conference on Management of Data*, pp. 227–238, 2016. https://doi.org/10.1145/2882903.2903731.

[10] McHugh J., Abiteboul S., Goldman R., Quass D., Widom J.: Lore: A Database Management System for Semistructured Data, *ACM SIGMOD Record*, vol. 26(3), pp. 54–66, 1997. https://doi.org/10.1145/262762.262770.

[11] Nadkarni P.M., Marenco L., Chen R., Skoufos E., Shepherd G., Miller P.: Organization of Heterogeneous Scientific Data Using the EAV/CR Representation, *Journal of the American Medical Informatics Association*, vol. 6(6), pp. 478–493, 1999. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC61391/.

[12] Orend K.: *Analysis and Classification of NoSQL Databases and Evaluation of Their Ability to Replace an Object-Relational Persistence Layer*. Master's Thesis, Technische Universität München, 2010.

[13] Piech M., Marcjan R.: A new approach to storing dynamic data in relational databases using JSON, *Computer Science*, vol. 19(1), pp. 3–20, 2018. https://doi.org/10.7494/csci.2018.19.1.2505.

[14] Rys M.: XML and relational database management systems: inside Microsoft® SQL Server™ 2005. In: *SIGMOD'05 Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 958–962, 2005. https://doi.org/10.1145/1066157.1066301.

[15] Sleator D.D., Tarjan R.E.: A Data Structure for Dynamic Trees, *Journal of Computer and System Sciences*, vol. 26(3), pp. 362–391, 1983. https://doi.org/10.1016/0022-0000(83)90006-5.

[16] Tahara D., Diamond T., Abadi D.J.: Sinew: A SQL System for Multi-Structured Data. In: *SIGMOD'14 Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 815–826, 2014. https://doi.org/10.1145/2588555.2612183.

[17] Tatarinov I., Viglas S.D., Beyer K., Shanmugasundaram J., Shekita E., Zhang C.: Storing and Querying Ordered XML Using a Relational Database System. In: *SIGMOD'02 Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 204–215, 2002. https://doi.org/10.1145/564691.564715.

[18] Whang K.Y., Park B.K., Han W.S., Lee Y.K.: Inverted index storage structure using subindexes and large objects for tight coupling of information retrieval with database management systems, US Patent US6349308, 2002. https://patents.google.com/patent/US6349308.

Affiliations

Mateusz Piech
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland, mpiech@agh.edu.pl, ORCID ID: https://orcid.org/0000-0002-0146-5921

Wojciech Frącz
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland, fracz@agh.edu.pl, ORCID ID: https://orcid.org/0000-0002-3613-6335

Wojciech Turek
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland, wojciech.turek@agh.edu.pl

Marek Kisiel-Dorohinicki
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland, doroh@agh.edu.pl, ORCID ID: https://orcid.org/0000-0002-8459-1877

Jacek Dajda
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland, dajda@agh.edu.pl

Aleksander Byrski
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, al. A. Mickiewicza 30, 30-059 Krakow, Poland, olekb@agh.edu.pl, ORCID ID: https://orcid.org/0000-0001-6317-7012

Received: 07.08.2018
Revised: 31.10.2018
Accepted: 31.10.2018