Computer Science • 19(1) 2018 https://doi.org/10.7494/csci.2018.19.1.2505

Mateusz Piech
Robert Marcjan

A NEW APPROACH TO STORING DYNAMIC DATA IN RELATIONAL DATABASES USING JSON

Abstract *JavaScript Object Notation was originally designed to transfer data; however, it soon found another use as a way of persisting data in NoSQL databases. Recently, the most-popular relational databases have introduced JSON as a native column type, which makes it easier to store and query dynamic database schema. In this paper, we review the currently popular techniques of storing data with a dynamic model with a large number of relationships between entities in relational databases. We focus on creating a simple dynamic schema with JSON in the most-popular relational databases, and we compare it with the well-known EAV data model and the document database. The results of precisely selected tests in the field of criminal data suggest that the use of JSON in dynamic database schema greatly simplifies queries and reduces their execution time compared to the widely used approaches.*

Keywords JSON, relation databases, EAV, criminal data, PostgreSQL

Citation Computer Science 19(1) 2018: 3–20

1. Introduction

Storing data with a dynamic structure in relational databases is not a trivial problem – these databases are well-suited for structured data with lots of relationships. While the most-popular way of persisting data with open schema is using a document database, sometimes it is not possible due to the need for ACID transactions or highly relational types of queries. There are also data models that allow for storing this kind of data in relational databases, such as Entity-Attribute-Value [21] or Inverted Index [27]; however, their main weaknesses is their complexity in preparing queries, execution time, and model readability. Our goal was to develop a new data model to store data with a dynamic structure in the relational database with query performance comparable to that of document databases and a structure simpler and more readable than in Entity-Attribute-Value.

Improving the open schema data model has an impact on several fields of science, such as medicine (clinical databases) or forensic science (criminal data), where the numbers of the types and attributes of objects are enormous and feature a lot of relationships between them. Inspiration for our research was closing the gap between relational databases and NoSQL databases through introducing JSON as a native type [16]. This facilitated a new approach to storing data with a dynamic schema.

The built-in support for the JSON type depends on the database engine. The first relational database management system that introduced native support for JSON was PostgreSQL [24], followed by MySQL [20] and Oracle Database [11]. Other databases like Microsoft SQL Server [23] provide only a library to handle JSON stored in the text column.

The inspiration for our work was from two issues. The first was the problem of optimal data storage and analysis of dynamic multi-domain criminal data [5]. Storing and processing dynamic data is prevalent in many other domains as well, such as clinical databases [3] or biomedical databases [21]. These databases have been using well-known and mature Entity-Attribute-Value representation to organize data. Our second inspiration was finding and testing a better data model. We decided to use a JSON document to store open schema data, inspired by the Sinew system.

This paper presents our open schema data model in a relational database. The model was tested by using carefully selected use cases in the field of criminal data. It was also compared with the Entity-Attribute-Value data model and MongoDB as the most-popular representative of document database engines. The results of our research show that using JSON to create a new dynamic data model simplifies and improves the current well-known data models, and it can replace current solutions.

2. Related work

Before databases adopted JSON, systems with an open schema in relational databases usually used the Entity-Attribute-Value or Inverted Index approaches. However, these techniques reduce the readability of entities and require a lot of work to process

complex queries, which results in longer execution times as compared to JSON. Most relational databases provide the ability to store XML as a native type, but this approach requires storing schema as well, resulting in its perception as a heavyweight format. The lack of satisfactory techniques for storing open schema initiated the attempts to create a proper solution using JSON.

One of the first interesting ideas was Argo [1] – an automated mapping layer for storing and querying JSON data in a relational system. One of its advantages was the ability to represent nested objects in a schema. The main concept was to store data in one big table with object id, keys, and three columns for string, number, or boolean, or in three tables with separate types (two variants of the approach). It was implemented in two relational databases (PostgreSQL, MySQL) and compared with MongoDB using the NoBench benchmark suite specifically designed for this purpose. Results show that Argo is faster than MongoDB for small data (1 million objects) but slower for bigger data (more than 16 million objects). While the concept was smart, it was only a slight improvement on the EAV data model, so the performance was not significantly improved.

Sinew is a system for storing documents of key-value pairs in relational database [26]. It was also tested on NoBench and compared with MongoDB, EAV, and Postgres with JSON. It is worth noting that, in these tests, Postgres with JSON outperforms EAV.

In the paper describing support for JSON as a native type in the Oracle Database [15], the Oracle Corporation presented results suggesting that indexed JSON is faster than competing solutions. Their next article concerned closing the functional and performance gap between SQL and NoSQL [16] and introduced OSON – a new query-friendly JSON format, which will be released in Oracle Database 12c Release 2.

3. Background

3.1. JSON

JavaScript Object Notification (JSON) [2] is a common data-interchange format designed primarily to serialize and transmit data over a network. It is used, for example, in the exchange of data between a client and server, REST-based services, and storing data in a NoSQL [9] database – MongoDB [4]. JSON has many advantages: it is relatively lightweight (when compared to alternatives such as XML), easy for humans to read and write as a text format, and has good support in many popular programming languages.

A JSON object consists of key-value pairs, where the key is a string and the value can be any JSON data type: string, number, another JSON object, or array – see example in Figure 1.

One of the extensions of this format is JSONB introduced by PostgreSQL. It allows for storing JSON as a binary object as opposed to a raw string. The purpose of such an approach is to facilitate the use of indexes and handle objects without the

need for parsing strings. A summary of the advantages of introducing binary JSON format in PostgreSQL (including a performance comparison) can be found in [14].

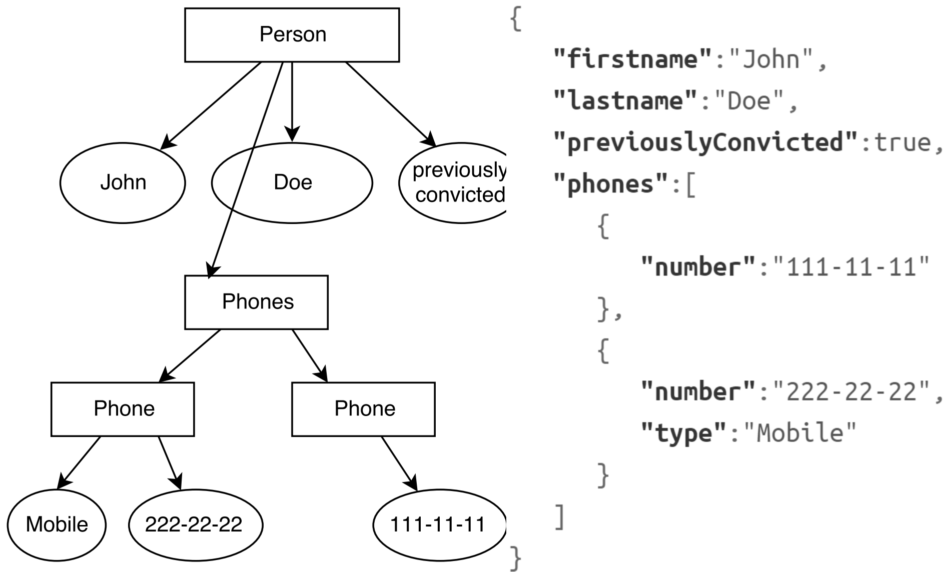

Figure 1. Graphical and textual representation of JSON object presenting person details

3.2. Criminal data

Crime analysis requires collecting data from the real world that contains many details related to the case – the so-called criminal intelligence data [25]. The data consist of two kinds of entities – objects (such as people, items or places, and relationships between these objects) and events (personal relationships, transactions, and other forms of association). Analysis of such data can often shed some light on the case.

An example of the graphic visualization of criminal data is presented in Figure 2, where the objects are represented by figures and relationships by connections. It depicts the heterogeneity of data – it is impossible to predict *a priori* the types of objects and relationships in a particular case, as they depend on the information collected during the investigation. For example, a person is involved in quite a few different types of relationships, even in this small example graph. Similarly, the same kind of relationship, like use, can denote an association between different kinds of objects. All of that suggests that the traditional approach used in relational databases (with a fixed schema designed upfront) is poorly suited for storing this kind of data.

The creation of schema for criminal data is a difficult problem due to the variety and unpredictable growth of the number of objects and relationships. Let us assume that an analyst examines bank statements to find money laundering. To store the data, he needs three tables in a database: one for sources and destinations of the

money, one for bank accounts, and one for money transfers. But then, the analyst gets another data set from an Internet service providing a virtual wallet. This forces him to extend the model with another pair of tables. Next, he gets a list of postal orders, which again must be included in the schema. The analyst can get more and more various data, which makes it difficult to prepare a complete schema when there is no specification of models and relationships for analysis.

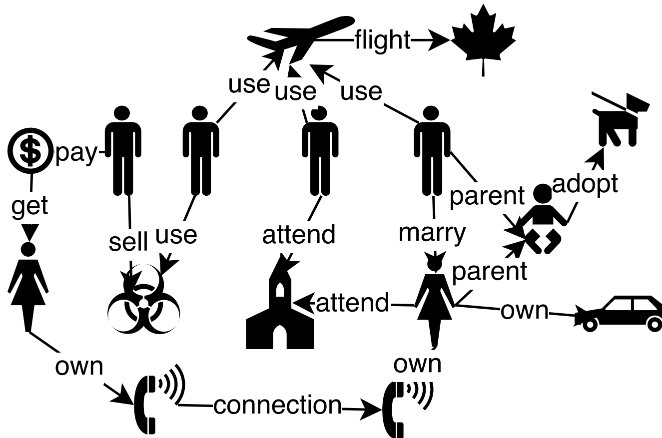

Figure 2. Representation of Criminal Data with phone connection between people and relationships between them

Even if the schema could be fully specified, the number of possible attributes for objects and relationships is likely to be large, and the actual data is often sparse – for each entry, only a small subset of all possible attributes is available. This necessitates the use of techniques that can handle sparse data efficiently, like EAV, document databases, or relational databases supporting JSON.

3.3. Relational database overview

We started the selection of relational databases suitable for our solution from doing a survey of the most-popular databases according to DB-Engines Ranking [6]. We chose four engines divided into two groups. The first group consists of open-source relational databases PostgreSQL and MySQL, in which we implemented our solution. The second consists of commercial engines Oracle Database and Microsoft SQL Server. Due to licensing restrictions, we did not include these in our tests [17, 22].

3.3.1. PostgreSQL

PostgreSQL [18] added support for JSON in Version 9.2 in 2012. The ability to store JSON as a native type resulted in recognizing PostgreSQL as Document Database [12].

In Version 9.6, there is a possibility to store JSON documents in JSONB format, which improves performance of querying due to using indexes and a binary format.

The primary operator is `->`, which retrieves JSON object by key. Nested objects are retrieved using operator `#>` with path in curly brackets, like `{address,home}`. Documents stored in JSONB format support additional operators: it is possible to check for the presence of a key in an object, concatenate objects and delete fields. A few functions to create and process JSON are provided as well.

3.3.2. MySql

MySQL [19] introduced support for native JSON in Version 5.7 [20]. MySQL supports handling JSON object with only one operator `->`, which is a reference to the extract value function. It allows for creating, editing, and processing JSON. Access to the nested objects is possible by providing a path that begins with a dollar sign; e.g., `$.address.home`.

The engine allows for creating indexes on JSON documents, but it requires the creation of a generated column storing values extracted from JSON objects by a given path. Unfortunately, this solution is not allowed in our implementation due to restrictions on modifying the model.

3.3.3. Microsoft SQL Server

Microsoft SQL Server [7] (MS SQL) added support for built-in JSON in Version 13.00 (SQL Server 2016). This is not a native support, because JSON is represented as a `NVARCHAR` type. MS SQL provides four operations on JSON objects: testing if a string is valid JSON, extracting value from JSON, querying a JSON object, and modifying it. It is possible to use indexes, but the solution is the same as for MySQL.

3.3.4. Oracle Database

Oracle Database [8] introduced support for native JSON in Version Oracle Database 12c Release 1. The function library to handle JSON is almost complete; the only missing feature is modifying a JSON document, which hinders the creation of a dynamic model. It forces us to update an entire column instead of the document field. Just like in PostgreSQL, it is possible to create indexes on a document.

4. Dynamic JSON model schema

Preparing a new open schema model requires an analysis of current models in order to select their advantages and improve their imperfections. In our research, we examined the Entity-Attribute-Value data model and technique of storing data in MongoDB. We mainly focused on the way the relationships are stored in these solutions, because it is the main issue in the dynamic model.

In the EAV model (Figure 3), it is hard to store dynamic objects with relationships. The id and type of the object are stored in the `entity` table, and they are used to find all related attributes and associated values from the table `value`. The relationships are stored in a separate table `entity_relation`, which contains information about the involved objects (`left`, `right`) and type of relationship (`relation_type`). However, if we want to store the attributes of the relationship itself, we have to represent relationships as entities and introduce another level of indirection by having `entity_relation` store only the IDs of these entities. From this open schema, we used dictionary definitions of the object and relationship types. Additionally, we moved ID to a table with the JSON document. We decided that an associative table with the relationships between entities used in EAV/CR almost fulfills its assumption, so we adapted it to the JSON model, adding a structure to store dynamic attributes.

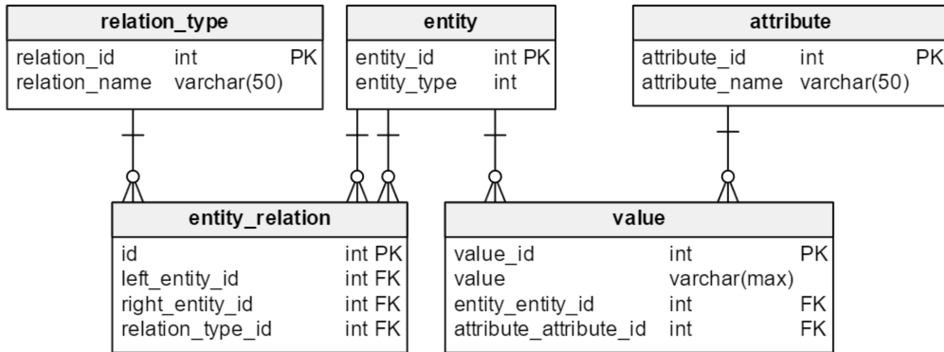

Figure 3. Schema of EAV model used in tests

MongoDB stores JSON documents in collections that are equivalent to a table in a relational database. In the documents, there are two ways of storing related objects. The first is creating embedded documents with an associated key. This is not suitable for our model because it creates duplicate data, making it difficult to provide transactions. The second is creating a reference to another document by ID (similar to using foreign keys in a relational database). Analysis of MongoDB has shown that storing dynamic objects in one table requires a way to distinguish between different types of a model. Our solution also requires a technique to link models by reference with the possibility of using foreign keys.

Our model is presented in Figure 4, and it consists of four tables. The first one is `Object_Name`, which contains a dictionary of the names of the criminal data objects. The second one – `Object` – is the main table, and it has columns that hold JSON documents, object IDs, and references to the type. The documents do not contain references to other documents like in MongoDB, because we have moved them to a third table – `Relations`, which stores the relationships between the criminal data objects. Apart from the two objects being related, each entry has an arbitrary list

of properties represented as a JSON document. The last table, `Relation_Name`, is similar to the first one and contains a dictionary of the names of the relationships present in the criminal data.

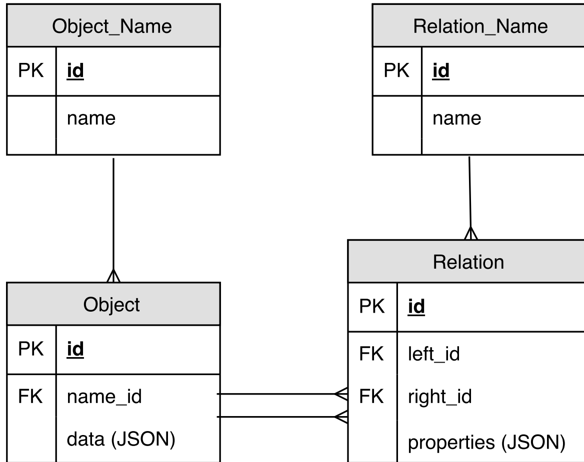

Figure 4. Schema of our new database model

4.1. Simple and readable model

One of our aims was to create a simpler and more-readable model than EAV. To test the intended result, we have prepared a simple query based on Figure 1. We wanted to find all persons filtered by three attributes. Its implementation is presented in Table 1. Comparing the realizations in the JSON and MongoDB models shows that both present a similar level of complexity. Additionally, the query in the JSON model looks almost the same as an analogous query using a table with static columns – the only differences is the extraction operator and the lack of type control.

The same query in the EAV model requires a full separate query for each part of the filter condition, which causes a significant bloat in the query code. Furthermore, this code is nothing but straightforward; and unlike other models, its semantics are buried beneath the implementation details – subqueries, joins, etc. It is worth noting that the EAV query in Table 1 returns only a list of object ids, not a list of objects themselves like in the other queries – this has been simplified for presentation purposes.

The difference in readability is clearly visible even in such a simple query; this will only be greater in more-complex cases. This is mainly due to the different ways of accessing the attributes, since the relationships between the objects are represented similarly in each approach. If there is a need to filter by relationship properties or associated object attributes, the EAV model requires a subquery to select the corresponding entities.

Table 1

Example query about person implemented in different techniques

Model	Query
JSON in relational database	SELECT * FROM object AS person JOIN object_name AS person_name ON person_name.id = person.name_id AND person_name.name = 'Person' WHERE person.data -» 'lastname' = 'Doe' AND person.data -» 'age' > 25 AND person.data -» 'previouslyConvicted' IS TRUE;
MongoDB	db.Person.find({$and: [{"lastname": {$eq: "Doe"}}, {"age": {$gt: 25}}, {"previouslyConvicted": true},]})
EAV	SELECT subquery.entity_id FROM (SELECT entity_id, 1 AS filter FROM value JOIN attribute ON value.attribute_id = attribute.attribute_id AND attribute.attribute_name = 'lastname' WHERE value.value = 'Doe' UNION ALL SELECT entity_id, 2 AS filter FROM value JOIN attribute ON value.attribute_id = attribute.attribute_id AND attribute.attribute_name = 'age' WHERE value.value > 25 UNION ALL SELECT entity_id, 3 AS filter FROM value JOIN attribute ON value.attribute_id = attribute.attribute_id AND attribute.attribute_name = 'previouslyConvicted' WHERE value.value IS TRUE) AS subquery JOIN entity ON subquery.entity_id = entity.entity_id AND entity.entity_name = 'Person' GROUP BY subquery.entity_id HAVING COUNT(subquery.filter) = 3;

5. Model evaluation

The last stage of our research was an evaluation of the developed model. We tested it by simulating real-use cases in the field of criminal data. For this purpose, we generated many datasets and selected four queries (presented for PostgreSQL with JSON in Table 4) with different levels of complexity to precisely examine all characteristics

of the models. The first query is a simple search for objects by attribute. The second and third use aggregate operations on a filtered set of objects. The last one heavily utilizes joins. The expected result was a performance improvement over the EAV solution.

We compared our solution implemented in two open-source relational databases: PostgreSQL and MySql, with the Entity-Attribute-Value data model and document database MongoDB. The testing computer had a 3.2Ghz quad-core Intel i5-4460 processor with 16 GB of memory and 256 GB of solid-state storage. We executed all queries ten times on each dataset; for each dataset, we selected the median of the results in order to compute the speedup. Below, we present the system configuration.

PostgreSQL with JSON

Starting with Version 9.5, PostgreSQL includes support for storing JSON in binary type. This feature allowed us to improve performance by creating indexes. We created indexes on all JSON elements used in queries and on all foreign keys to speed up join operations. In our tests, we used PostgreSQL 9.6.1 installed with the default settings.

MySQL with JSON

We installed MySQL Community Server 5.7.16 with the default settings to compare the prepared model in a different relational database system. Unfortunately, there is no possibility to create indexes on JSON documents due to issues mentioned before, so we created indexes only for foreign keys.

Entity-Attribute-Value in PostgreSQL

To compare our prepared model with the Entity-Attribute-Value model, we implemented this solution in PostgreSQL (see Figure 3) – in the same instance as in the JSON model. In the EAV model, there is no separation between the object and relationship in the criminal data domain, so we represented both as entities. We also created indexes on foreign keys, like in the other solutions.

MongoDB

MongoDB is the most-popular document database, so comparing it with our solution objectively determines if storing dynamic data as JSON in relational databases is an efficient approach. In our tests, we used Community Server 3.2.11 installed with the default settings. We stored objects and relationships in separate collections and linked them using references to objects.

5.1. Data sets description

The prepared datasets are comprised of criminal data representing phone billings. This kind of data includes not only people, phones, and connections, but also many other objects and relationships – seemingly irrelevant, but important to simulate the real-world context of the investigated case. For this purpose, we selected extra

object and relationship types. Every entity had a few properties of different types, including nested objects. We generated datasets for all combinations of size (100K, 250K, 500K, 750K, 1M, 2.5M, 5M, 7.5M, 10M) and five different subsets of objects and relationships.

Additionally, we measured the time and space required to insert and store the data from the dataset having the same object and relationship types and the most-representative sizes (100K, 1M, 10M). The results are presented in Tables 3 and 2, respectively. The results suggest that our model is scalable with increasing data size. The difference of size between PostgreSQL and MySQL is mainly due to the possibility of storing indexes.

Table 2

Size of datasets on disk in different engines

Database	100K records	1M records	10M records
PostgreSQL with JSON	23 MB	221 MB	2216 MB
MySQL	20 MB	152 MB	1507 MB
PostgreSQL with EAV	35 MB	395 MB	3225 MB
MongoDB	7 MB	74 MB	739 MB

Table 3

Time of populate datasets into databases

Database	100K records	1M records	10M records
PostgreSQL with JSON	14 s	150 s	2020 s
MySQL	24 s	229 s	2383 s
PostgreSQL with EAV	30s	314 s	3899 s
MongoDB	13 s	131s	1336 s

5.2. Query tests

To examine the performance differences between our approach and the Entity-Attribute-Value data model, we prepared four analytical queries designed to simulate plausible real-life use cases in the field of criminal data analysis. The implementation of the queries in the proposed model is presented in Table 4. We executed queries for each solution ten times for each dataset. Then, we used medians of the execution times for each dataset to compute the speedup relative to the Entity-Attribute-Value model. Finally, we computed median speedups for fixed data sizes. The median speedups are presented in Figure 5 and Table 5. We also included results for the most-complex dataset (by number of different objects and relationships) in Table 6.

Table 4
Queries implemented in our model used in tests

ID	Query
Q1	SELECT DISTINCT data-»'Lastname' FROM object JOIN object_name on object.name_id = object_name.id ÂND object_name.name = 'Person'
Q2	SELECT AVG(properties-»'Length') FROM relation JOIN relation_name ON relation_name.id = relation.name_id AND relation_name.name = 'Connection' WHERE properties-»'Date' BETWEEN '2016-03-01' AND '2016-05-31'
Q3	SELECT left_id FROM relation JOIN relation_name ON relation_name.id = relation.name_id AND relation_name.name = 'Connection' GROUP BY left_id HAVING COUNT(left_id) >1
Q4	SLECT DISTINCT caller.id, receiver.id FROM relation AS connection JOIN relation_name AS connection_relation ON connection.name_id = connection_relation.id AND connection_relation.name = 'Connection' JOIN object AS phone_caller ON connection.left_id = phone_caller.id JOIN object AS phone_receiver ON connection.right_id = phone_receiver.id JOIN relation AS phone_caller_owner ON phone_caller_owner.left_id = phone_caller.id JOIN relation_name AS phone_caller_relation ON phone_caller_owner.name_id = phone_caller_relation.id AND phone_caller_relation.name = 'Owning' JOIN relation AS phone_receiver_owner ON phone_receiver_owner.left_id = phone_receiver.id JOIN relation_name AS phone_receiver_relation ON phone_receiver_owner.name_id = phone_receiver_relation.id AND phone_receiver_relation.name = 'Owning' JOIN object AS caller ON phone_caller_owner.right_id = caller.id JOIN object AS receiver ON phone_receiver_owner.right_id = receiver.id WHERE connection.properties -»'Length' >120

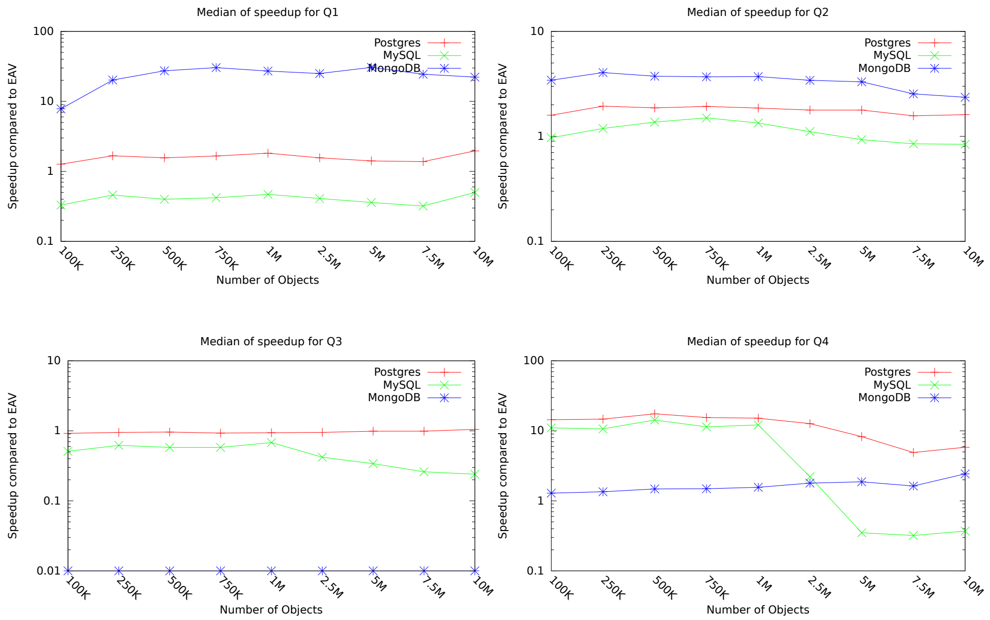

Figure 5. Median of speedup obtained for test queries during tests

Table 5

Median of speedup obtained for test queries during tests

	PostgreSQL				MySQL				MongoDB			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
100K	1.27	1.59	0.92	14.42	0.33	0.97	0.51	11.02	7.83	3.43	0.01	1.29
250K	1.67	1.94	0.95	14.73	0.46	1.19	0.62	10.68	20.17	4.05	0.01	1.35
500K	1.56	1.87	0.96	17.52	0.40	1.37	0.58	14.30	27.38	3.74	0.01	1.48
750K	1.66	1.93	0.93	15.49	0.42	1.50	0.58	11.41	30.33	3.69	0.01	1.49
1M	1.82	1.86	0.94	15.15	0.47	1.34	0.68	12.16	27.13	3.71	0.01	1.56
2.5M	1.56	1.78	0.95	12.68	0.41	1.11	0.42	2.21	24.98	3.42	0.01	1.79
5M	1.41	1.78	0.99	8.26	0.36	0.93	0.34	0.35	30.64	3.31	0.01	1.87
7.5M	1.38	1.57	0.99	4.91	0.32	0.85	0.26	0.32	24.45	2.54	0.01	1.63
10M	1.96	1.61	1.05	5.83	0.50	0.84	0.24	0.37	22.21	2.36	0.01	2.43

Table 6
Speedup and time obtained for most-complex dataset during tests

		PostgreSQL				MySQL				EAV in PostgreSQL				MongoDB			
		Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
100K	Time [ms]	2	3	1	28	16	15	4	47	5	7	1	601	2	3	54	306
	Speedup	2.50	2.33	1.00	21.46	0.31	0.47	0.25	12.79	1.00	1.00	1.00	1.00	2.50	2.33	0.02	1.96
250K	Time [ms]	4	8	4	81	47	16	16	125	13	19	3	1577	2	4	309	765
	Speedup	2.60	2.38	0.75	19.57	0.42	1.19	0.19	12.68	1.00	1.00	1.00	1.00	6.50	4.75	0.01	2.07
500K	Time [ms]	10	18	9	159	78	31	16	172	26	41	8	3538	2	10	1276	1638
	Speedup	2.60	2.28	0.89	22.25	0.33	1.32	0.50	20.57	1.00	1.00	1.00	1.00	13.00	4.10	0.01	2.16
750K	Time [ms]	15	28	14	241	109	31	31	328	42	58	12	4950	2	13	1702	2272
	Speedup	2.80	2.07	0.86	20.54	0.39	1.87	0.39	15.09	1.00	1.00	1.00	1.00	21.00	4.46	0.01	2.18
1M	Time [ms]	21	43	20	350	125	47	31	375	57	80	18	6948	10	20	5067	3241
	Speedup	2.71	1.86	0.90	19.85	0.46	1.70	0.58	18.53	1.00	1.00	1.00	1.00	5.70	4.00	0.00	2.14
2.5M	Time [ms]	54	91	47	838	297	125	109	1125	458	196	45	18553	6	50	28923	7940
	Speedup	8.48	2.15	0.96	22.14	1.54	1.57	0.41	16.49	1.00	1.00	1.00	1.00	76.33	3.92	0.00	2.34
5M	Time [ms]	110	181	101	2493	719	453	282	109750	673	431	103	38380	20	111	75116	16238
	Speedup	6.12	2.38	1.02	15.40	0.94	0.95	0.37	0.35	1.00	1.00	1.00	1.00	33.65	3.88	0.00	2.36
7.5M	Time [ms]	176	272	153	6044	1141	652	484	150094	851	641	154	60300	33	174	115372	23776
	Speedup	4.84	2.36	1.01	9.98	0.75	0.98	0.32	0.40	1.00	1.00	1.00	1.00	25.79	3.68	0.00	2.54
10M	Time [ms]	237	379	215	10147	1594	937	583	506866	1176	897	286	90036	40	278	182265	33846
	Speedup	4.96	2.37	1.33	8.87	0.74	0.96	0.49	0.18	1.00	1.00	1.00	1.00	29.40	3.23	0.00	2.66

First query

The first query consists of searching objects by the value of a specified attribute. The purpose of this test is to examine the performance of access to the dynamic attributes. The query selects all unique last names of people in the datasets. We can observe that the median speedups for all of the engines are at a similar level. However, our model does not achieve the performance of MongoDB. We achieve the highest speedup for datasets with the largest number of attributes. For the dataset with the smallest number of unique attributes (which prevents the efficient use of indexes), we observed the lowest speedup (below 1). This indicates that our solution provides the best performance benefits in the presence of complex data.

We noticed that the MySQL solution is slower than the Entity-Attribute-Value model. This is caused by the low performance of MySQL compared to PostgreSQL, which has been addressed in [10, 13].

Second query

The second query is of the report type and returns the average time of the connections in a specified time period. This allows us to examine the performance of the aggregation functions for the objects within the dynamic data model. Again, we can observe that the median speedup for each engine is similar and decreases slightly with increasing problem size. This can be explained by the cost of accessing JSON decreasing relative to the cost of aggregation.The results show that speedup does not depend on the type of dataset and that our solution approaches the level of MongoDB's performance.

Third query

The third test involves a more-complex report type query, utilizing aggregation, join, and grouping – it selects people that started the call more than once. For this query, MongoDB is significantly slower than our model and the reference EAV model. This seems to suggest that MongoDB performs poorly in the presence of filtering based on conditions involving aggregate functions and joins. However, the PostgreSQL solution and EAV have almost the same performance, as both solutions have a very similar execution plan. It is completely understandable because, in this test, we did not access the dynamic data. As with the first query, we observe the performance issues of MySQL.

Fourth query

The final query was designed to simulate a simple graph query, searching for pairs of people who communicated with each other for longer than a specified amount of time. It requires a lot of joins, so it evaluates the performance of queries involving the relationships of objects. In this test, our model has the highest speedup among the tested solutions and the highest overall speedup observed in the entire experiment. These results prove that the Entity-Attribute-Value model is too complex and MongoDB is not a good engine to execute queries involving relationships. However, there

is a decreasing trend for larger data sizes. This is due to the fact that, for a large-enough data set size, there is a need to use the disk, because the space required for the operation exceeds the available memory, which is marked in the execution plans.

5.3. Discussion

The conducted tests demonstrate that our solution in PostgreSQL is, in general, a little faster than Entity-Attribute-Value and much faster in the case of complex queries and datasets. Furthermore, it leads to simpler schema than the Entity-Attribute--Value and can be used in the field of criminal data, as well as other fields that require a dynamic data model with relationships.

Our main goal was to check whether our model is faster than the Entity--Attribute-Value. The most-significant results are those concerning the model implemented in PostgreSQL, which is also the engine in which the Entity-Attribute-Value solution was implemented. For all queries, the obtained speedup is almost constant with respect to the size of the dataset; the only increasing trend is observed when there is a need to use disk space to execute a query.

We achieved all set goals by representing dynamic data in the JSON document. It resulted in the improvement of readability and simplifying the creation of queries by reducing the numbers of joins in graph queries.

While the observed performance of the implementations of our solution in PostgreSQL and MySQL differ significantly in some cases, the results remain acceptable considering MySQL's performance issues. This suggests that the speedup provided by our solution (relative to the EAV model) is not dependent on a particular database engine as long as it provides JSON support.

The test results for MongoDB show that, while a document database is indeed the best solution for storing and retrieving data from a dynamic model, the same is not true when use cases are more complicated and involve aggregation or relationships. For these types of queries, MongoDB clearly fails to exhibit performance comparable with our model.

6. Conclusion and future work

In this paper, we presented a solution for storing open schema data in a relational database supporting JSON as a native type. The prepared model was set in the field of criminal data, but it can be easily adapted to other domains. Our solution has been implemented in two relational databases and compared with the NoSQL document database – MongoDB, and the Entity-Attribute-Value solution commonly used in relational databases. The results of the conducted tests confirm that storing data with dynamic schema in a relational database using JSON can be even more efficient than MongoDB in some cases.

In the future, we want to focus on developing the presented model and adapting it to graph queries, which are prevalent in the analysis of criminal data. This will allow

us to implement complex algorithms for data queries like searching paths or closest neighbors and, as a result, bring us one step closer to creating a general solution with the benefits of NoSQL databases.

Acknowledgements

This research was partially supported by Grant No. DOB-BIO6/08/129/2014 from the Polish National Center for Research and Development.

References

[1] Chasseur C., Li Y., Patel J.M.: Enabling JSON Document Stores in Relational Systems. In: Proceedings of 16th International Workshop on the Web and Databases (WebDB 2013), pp. 1–16, 2013.

[2] Chen H.: Javascript object notation schema definition language, US Patent App. 13/596,694, 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.

[4] Chodorow K.: *MongoDB: the definitive guide*, O'Reilly Media Inc., 2013.

[5] Dajda J., Dębski R., Kisiel-Dorohinicki M., Piętak K.: Multi-domain data integration for criminal intelligence. In: *Man-Machine Interactions 3*, pp. 345–352. Springer, 2014.

[6] DB-Engines Ranking. http://db-engines.com/en/ranking.

[7] Gray J.: Microsoft SQL Server, 1997. https://www.microsoft.com/en-us/research/publication/microsoft-sql-server/.

[8] Greenwald R., Stackowiak R., Stern J.: *Oracle essentials: Oracle database 12c*, O'Reilly Media Inc., 2013.

[9] Han J., Haihong E., Le G., Du J.: Survey on NoSQL database. In: *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pp. 363–366, IEEE, 2011.

[10] Jajeśnica Ł., Piórkowski A.: Productivity and nesting join the schemes and standarized and denormalized, *Studia Informatica*, vol. 31(2A), pp. 445–456, 2010.

[11] JSON in Oracle Database. https://docs.oracle.com/database/121/ADXDB/json.htm.

[12] Lerner R.M.: At the forge: PostgreSQL, the NoSQL database, *Linux Journal*, vol. 2014(247), p. 5, 2014.

[13] Lim N.H.: PostgreSQL [9.5.0] vs MariaDB [10.1.11] vs MySQL [5.7.0] year 2016. http://nghenglim.github.io/PostgreSQL-9.5.0-vs-MariaDB-10.1.11-vs-MySQL-5.7.0-year-2016/.

[14] Litt G., Thompson S., Whittaker J.: *Improving performance of schemaless document storage in PostgreSQL using BSON*, CPSC 438 Final Project, April 29, 2013, New Haven, CT, 2013.

[15] Liu Z.H., Hammerschmidt B., McMahon D.: JSON data management: supporting schemaless development in RDBMS. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 1247–1258, 2014.

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

[17] Microsoft SQL Server End-User License Agreement. `http://contracts.onecle.com/aristotle-international/microsoft-eula.shtml`.

[18] Momjian B.: *PostgreSQL: introduction and concepts*, vol. 192, Addison-Wesley, New York, 2001.

[19] MySQL A.: MySQL, 2001.

[20] MySQL – The JSON Data Type. `https://dev.mysql.com/doc/refman/5.7/en/json.html`.

[21] 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.

[22] Oracle Technology Network License Agreement. `http://www.oracle.com/technetwork/licenses/standard-license-152015.html`.

[23] Popovic J.: JSON Support in SQL Server 2016. `https://blogs.msdn.microsoft.com/jocapc/2015/05/16/json-support-in-sql-server-2016/`.

[24] PostgreSQL – JSON Types. `https://www.postgresql.org/docs/9.6/static/datatype-json.html`.

[25] Sparrow M.K.: The application of network analysis to criminal intelligence: An assessment of the prospects, *Social Networks*, vol. 13(3), pp. 251–274, 1991.

[26] Tahara D., Diamond T., Abadi D.J.: Sinew: a SQL system for multi-structured data. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 815–826, 2014.

[27] 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*, 2002. US Patent 6,349,308.

Affiliations

Mateusz Piech
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, Krakow, Poland, mpiech@agh.edu.pl

Robert Marcjan
AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Department of Computer Science, Krakow, Poland, marcjan@agh.edu.pl

Received: 08.05.2017
Revised: 18.12.2017
Accepted: 18.12.2017