On the performance impact of using JSON, beyond impedance mismatch

Moditha Hewasinghage, Sergi Nadal, and Alberto Abelló

Universitat Politècnica de Catalunya (BarcelonaTech), Barcelona, Spain
{moditha,snadal,aabello}@essi.upc.edu

Abstract. NOSQL database management systems adopt semi-structured data models, such as JSON, to easily accommodate schema evolution and overcome the overhead generated from transforming internal structures to tabular data (i.e., impedance mismatch). There exist multiple, and equivalent, ways to physically represent semi-structured data, but there is a lack of evidence about the potential impact on space and query performance. In this paper, we embark on the task of quantifying that, precisely for document stores. We empirically compare multiple ways of representing semi-structured data, which allows us to derive a set of guidelines for efficient physical database design considering both JSON and relational options in the same palette.

1 Introduction

The relational model was defined as an abstraction level to gain independence of the file system and any internal storage structure [6]. Thus, we could gain flexibility and interoperability without losing efficiency by following a tabular representation and some normal forms. Indeed, the first normal form (1NF) established that attribute domains had to be atomic (i.e., they could be neither compound-complex structures nor arrays). However, a rigid tabular structure is not adequate in modern agile software development, where the schema is under continuous evolution. Moreover, a well-known problem of RDBMS is the impedance mismatch, defined as the overhead generated by transformations from internal structures to tables, and then into programming structures [3].

The development of NOSQL systems, which adopt more flexible data representations, allowed to overcome the impedance mismatch [14]. Such data formats (e.g., JSON), are directly mapped from disk to memory.This is additionally achieved by breaking 1NF, allowing typical programming nested structures and arrays in the attribute values (e.g., MongoDB encourages denormalization[1]). Furthermore, such semi-structured formats, also allow to skip schema declaration, which is beneficial in highly evolving applications [13]. Nevertheless, it is not clear whether denormalization and schemaless is a conscious design choice, or merely a paradigm imposed by the limitations of NOSQL systems. Yet, the

Partly funded by the European Commission through the programme “EM IT4BI-DC”. We thank Braulio Blanco for assisting on the first version of the experiments.

[1] https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-2

flexibility offered by NOSQL comes at a price, where each one of the associated design choices may widely change their physical representation, and thus profoundly impact performance. Practitioners have ignored this, and today make binary design decisions based on rules, and programming needs with no overall view of the system needs [12]. Thus, it is vital to consider the benefits and drawbacks posed by these different alternatives during the design process [5]. Relational and semi-structured data models, are not a simple binary choice, but a continuum of options with different degrees of (de)normalization.

In this paper, we quantify the performance impact of physical database design choices on NOSQL systems, focusing on the JSON data model. To this end, different design choices (i.e., equivalent representational differences) related to both metadata (i.e., schema), such as attribute embedding or optionality, and data, such as nested objects or arrays, are quantitatively scrutinized. We acknowledge that many DBMS features can affect performance (i.e., concurrency control and recoverability mechanism, distribution and parallelism management, connection pools and setup, etc.).Nevertheless, we only study the impact of design decisions on a semi-structured data model, being agnostic of the technological choice. Our main contributions are as follows: 1. We identify the main physical design characteristics of semi-structured data and compare them to their structured counterpart. 2. We empirically quantify the impact of design choices in semi-structured data. 3. We evaluate the different designs in a relational and NOSQL DBMS.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 presents design differences. Section 4 shows experimental results. Sections 5 and 6 discuss the experimental findings and conclude the paper.

2 Related Work

[4] abstracts and homogenizes the modeling commonalities of NOSQL systems. It considers databases as sets of collections, which in turn are sets of blocks, finally represented by sets of entries. Similarly, [9] proposes a subject-oriented methodology to design NOSQL databases. A conceptual model of the system is converted into an equivalent hypergraph representation, such that hyperedges identify specializations or aggregations among entities. For each hyperedge, an specific data model, either relational or co-relational. [7] proposes a method to generate NOSQL databases from a high-level conceptual model automatically. The authors propose the UML-like Generic Data Metamodel, integrating structural and data access patterns. Then, a set of transformation rules generate the specific constructs for the target model (e.g., document or column-family).

Regarding performance, [8] benchmarks PostgreSQL and MongoDB. An OLAP-like workload is evaluated in both systems on real-world data from Github. The benchmark concludes that PostgreSQL yields higher performance results, but different design alternatives are not explored. [11] explores the impact of normalized collections w.r.t. embedded objects in MongoDB, and empirically shows that querying embedded objects is orders of magnitude faster than their normalized counterpart using joins. Similarly, [15] benchmarks systems in the NOSQL realm (i.e., MongoDB and CouchDB) as well as RDBMSs with built-in JSON support (i.e., PostgreSQL and MySQL). This work differs from our setting, as it focuses on CRUD transactions for a simple document structure.

3 Representational Differences

The term semi-structured describes data that have some structure but is neither regular nor known a priori [1]. For example, a JSON document consists of a nested hierarchy of key-value pairs with a single root. Child documents are an unordered sequence list of pairs with optional presence. Hence, JSON documents are self-descriptive, and do not require a schema declaration, despite a known structure facilitates storage and encourages queries [2]. Conversely, a structured database distinguishes schema and instances. The former is a set of attributes, each with a concrete domain, while the later is a tuple of values that belong to the corresponding domain in the previously declared schema. Hence, here, we present representational differences between semi-structured and structured data (i.e., equivalent alternatives to represent some datum exploiting the characteristics offered by each of both models), and discuss their potential impact on storage size, data insertion, and query performance. For each representational difference, we present patterns used in the empirical validation in Section 4.

3.1 Schema variability

A common schema is defined for all instances in structured databases, but in JSON, there may exist potentially different document schemata inside the same collection. Here, we focus on comparing alternative ways to represent the schema.

3.1.1 Metadata representation

Representing different schemata across JSON documents entails embedding their metadata into each instance (Fig. 1). This clearly impacts negatively the size of the database and consequently query performance. The more attributes are present, the more metadata (i.e., attribute names) will be embedded into each document. Additionally, the ratio between the size of data and metadata is clearly an important factor to consider (i.e., attribute name length w.r.t. its values). Thus, we need to consider (a) the absolute amount of metadata by analysing different number of attributes (from 1 to n), and (b) the relative amount of metadata by analysing different ratios (by increasing the value length from 1 to m, while at the same time that decreases the attribute name length in the same number of characters).

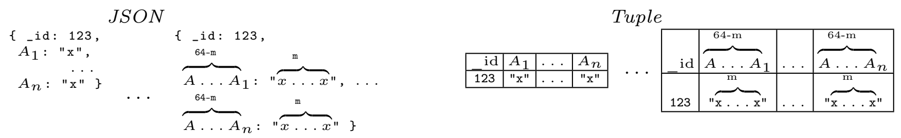


Fig. 1: Alternative representations of Metadata

3.1.2 Attribute optionality

Another feature of the semi-structured data model is the possibility to skip the representation on an attribute in the absence of its value (as the case of *J-Abs*, Fig. 2). However, it also supports to, either use a special value outside the domain (as in *J-NULL*) or use a specific value inside the attribute domain (as in *J-666*). Notice that in a relational representation, as the schema is fixed and common to all instances, only the last two options are possible (as in *T-NULL* and *T-666*, respectively). The impact on space and performance of these representations will vary depending on the percentage of absent/present values for the attribute.

J-Abs

```
{ _id: 123 }
```

J-NULL

```
{ _id: 123,
 A1: null, ...
 An: null }
```

J-666

```
{ _id: 123,
 A1: 666, ...
 An: 666 }
```

T-NULL

_id	A_1	...	A_n
123	null	...	null

T-666

_id	A_1	...	A_n
123	666	...	666

Fig. 2: Alternative representations for optional attributes

3.2 Schema declaration

In order to benefit from Schema declaration and validation in semi-structured databases, one must adopt additional constructs. `JSONSchema` is a JSON-based schema language that allows to constrain the shape, types and values of JSON documents. Here, we will evaluate the impact of both structure plus data type declaration, and integrity constraint (IC) validation separately.

3.2.1 Structure and data types

To validate structure and data types, `JSONSchema` uses the `properties` key. For each attribute, it is possible to specify its data type, which can be either a primitive or complex object. Furthermore, the `required` key represents an array enumerating the list of expected attributes. Fig. 3 depicts the exemplary document patterns considered. Clearly, this declaration has no impact on database size, since it does not grow with instances. However, it has a cost on insertion, corresponding to validating presence and domain, and on the other hand, it could potentially benefit query time by saving an explicit casting and type conversion.

J-Typ

```
{ _id: 123,        { "type": "object", "properties": {
 A1: k, ...          "A1": {"type": "number"}, ...
 A64: k }            "An": {"type": "number"},
                      required: ["A1",...,"An"] } }
```

T-Typ

_id	A_1	...	A_{64}
123	k	...	k

Fig. 3: Alternative representations of structure and data type validation

J-IC

```
{ _id: 123,   { "type": "object", "properties": {
 A1: k,         "A1": {
      ...        "type": "number",
 A64: k }        "minimum":-k′,"maximum: k′}, ...
                "An": {
                 "type": "number",
                 "minimum":-k′,"maximum: k′} }
```

T-IC

_id	A_1	...	A_{64}
123	k	...	k

```
ALTER TABLE T ADD CONSTRAINT
val_A1 CHECK
(A1 BETWEEN -k′ AND k′);
...
ALTER TABLE T ADD CONSTRAINT
val_An CHECK
(An BETWEEN -k′ AND k′);
```

Fig. 4: Alternative representations of Integrity Constraints (IC) validation

3.2.2 Integrity constraints

Besides the data type validation mechanisms, `JSONSchema` also offers means to represent integrity constraints for attributes. Here, as depicted in Fig. 4, we focus on enforcing ranges of values. In relational databases, this is achieved via `CHECK` constraints. As above, this has no impact on the size of the database but will have some on the insertion since it has to be checked before accepting the data. Despite this, it might also be used to perform some semantic optimization at query time; we consider this is technology-specific (i.e., not directly dependent on the data representation) and will not be evaluated in Section 4.

3.3 Structure complexity

An RDBMS conforms to 1NF, yet a semi-structured one relaxes such restriction, which allows storing nested and multi-valued data. Here, we study the impact of different complexity degrees on data according to that.

3.3.1 Nested structures

Documents allow to explicit into a data structure conceptually independent objects, which are accessed using dot notation. Yet, it is unclear what is the impact regarding size (i.e., with an increasing number of brackets in the document), and on querying such structures. To explore this, we will experiment with a range of levels and attributes (*Nest-one* and *Nest-all* in Fig. 5). Precisely, we will evaluate (a) increasing document sizes (i.e., *Nest-one*), and (b) constant document sizes (i.e., *Nest-all*); both w.r.t. the number of nesting levels. *Nest-1* indicates that there is only one attribute in the lowest level, while *Nest-all* contains less attributes the more levels we have. For instance, with 32 nesting levels, *Nest-one* has only A_{33}, while *Nest-all* has attributes A_{33} to A_{64}. Thus, in the latter, for every level we add together with the required extra characters (i.e., *:*, *{*, and *}*), we remove an attribute. Consequently, the overall size remains constant in terms of document length, but not in physical storage space due to the encoding of integer values being used.

Nest-one

```
{ _id: 123,
  L1: { ...
    Ln: {
      An+1: k }
            ...}}
```

Nest-all

```
{ _id: 123,
  L1: { ...
    Ln: {
      An+1: k, ...
      A64: k }...}}
```

J-Arr

```
{ _id: 123,
 A: [1,...,n]
        }
```

J-Att

```
{  _id:
        123,
 A1: k,
        ...
 An: k }
```

T-Arr

_id	A
123	[1,...,n]

T-Att

_id	A_1	...	A_n
123	k	...	k

Fig. 5: Representations of nesting structures and multi-valued attributes

3.3.2 Multi-valued attributes

Only modern object-relational DBMSs have adopted variable-length multidimensional arrays as data type, an aspect present in JSON by definition. Yet, it is unclear what is the impact of managing such types. On bounded arrays, one could argue that it might be better to store each position as an independent attribute, as depicted in Fig. 5, where we distinguish, for both JSON and tuples, *array* and *multi-attribute* alternatives. Multi-valued attributes could also be stored in a separate normalized table, however such independent structure would compete for resources, heavily impacting insertion and query [10]. We consider such eviction policies are technology-specific, thus they will not be evaluated.

4 Experimental evaluation

We conducted experiments to evaluate the choices discussed in Section 3, using PostgreSQL v12 (which supports native JSON storage) to compare the differences between relational and JSON alternatives. We also used MongoDB v4.2 (nowadays, the most popular document store) to validate the consistency of results. Note our objective is not to perform a technological comparison, but to evaluate the impact of document design choices. No specific tuning was performed for any system, using the default parameters. We disabled compression in MongoDB to facilitate its comparison with PostgreSQL, and cleared the operating system cache and restarted the DBMS between each execution to clear caches. We got three metrics: (a) storage size in MB;(b) overall runtime of insertions in seconds; and (c) median runtime to aggregate a numeric attribute in seconds over 20 repetitions. To store JSON in PostgreSQL, we created a table with two attributes: a `CHAR(24)` to store the ID (equivalent to `Object_ID` in MongoDB) and a `JSONB` to store the document. Then, we generated 1 million random documents according to each schema pattern in Section 3, over an exponentially increasing parameter, which were inserted in 100 batches of 10K documents. Due to space limits, we omit the individual figures[2]. To minimise impedance mismatch, queries return a single value aggregating numerical attributes. Note that MongoDB stores 32-bits integers[3], while PostgreSQL uses 64-bits[4], which in the end causes differences on storage size and consequently in insertion and query performance.

4.1 Schema variability

For schema variability, we conducted three experiments overall because we already had two patterns regarding metadata embedding (Section 3.1.1): (i) change the number of numeric attributes in a document; and (ii) change the data-metadata ratio, keeping a fixed number of attributes.

Varying document size. According to our experiments JSON always requires more space than tuples, due to metadata being replicated in every document. We can observe the same trend in insertion times. However, although storage space for a tuple is smaller in all cases, insertion time is shorter only for few (i.e., four) attributes. Beyond that, JSON insertion is faster (due to no type checking, as shown later in Section 4.2). At query time the runtime increases with the number of attributes. However, oppositely to insertion, tuples perform faster (since they benefit from the work done at insertion time). In all cases, we can see that PostgreSQL and MongoDB follow the same trend on storing JSON. They only differ in the physical format, which requires less space in the latter (64-bit vs. 32-bit integers). Thus, MongoDB generates less I/O (roughly half), improving insertion and query time.

Constant document size. Aiming to stabilise the overall size of the document, we keep constant the sum of characters between attribute name and value. Thus,

[2] Source code and all graphs available at `https://github.com/dtim-upc/MongoDBTests`

[3] `https://docs.mongodb.com/manual/reference/bson-types`

[4] `https://www.postgresql.org/docs/12/datatype-json.html`

we have one numerical attribute for the queries and consider nine other string attributes, changing at once their data to metadata ratio by changing the length of attribute name and value keeping a constant of 64 characters for both together. The number is chosen based on PostgreSQL having a limit of 63 characters for attribute names, so the attribute name length ranges from 1 to 63 and the value length from 63 to 1. Since attribute name is only stored once, independently of the number of tuples, the storage space taken by the tuples decreases with the growth of the attribute name length. Oppositely, attribute names are redundantly stored in all documents in JSON, so the overall size remains constant except for 63 characters, seemingly due to the presence of a step function in physical storage allocation. This is confirmed in MongoDB, where the gradual growth in space is more apparent. Interestingly, PostgreSQL and MongoDB storage size for JSON is much closer in this experiment as most of the attributes are strings instead of integers. Insertion and query times follow the same trend as the attribute length grows indicating I/O is always the dominant factor.
Optional attributes. Regarding attribute optionality, we consider five alternatives to represent the absence of values in the attributes (Section 3.1.2). Thus, the pattern consists of 64 integer attributes (potentially removed all at once), and one fixed-length string of size 64 to guarantee a minimal document size when the former are removed. Thus, we varied the percentage of documents without value for their integer attributes. Regarding storage space, the worst option to represent absence of data is using a value inside the domain (i.e., *T-666* and *J-666*), which keeps a constant size. In both tuples and JSON, we can use a *null* special value (i.e., *T-NULL* and *J-NULL*), which clearly saves space as attribute values disappear. However, the complete absence of the attribute in JSON (namely *J-Abs*), reduces the storage space the most due to the saving also in the metadata. As before, storage space in MongoDB follows the same trend as in PostgreSQL, but with smaller values due to the different encoding of integers. Regarding insertion time, the trend coincides with that of the storage used for JSON in both systems. However, tuples in PostgreSQL keep a constant insertion time, because the dominant factor is not I/O, but validation and formatting of data, which is not even compensated by the saving in metadata storage. When querying the data, we tested both summing and counting their presence with similar results. In all cases, the dominant factor of the query time is I/O, and consequently follows the trends and proportions of storage space.

4.2 Schema declaration

As discussed, schema declaration does neither affect the overall storage size nor query time. Thus, we measure insertion time for both data types and ICs.

Type and constraint validation. Regarding type and IC checking (Sections 3.2.1 and 3.2.2), we generated documents with 64 attributes and declared type and ICs in an incremental manner (from 1 to 64). To enforce JSON schema declaration in PostgreSQL, we used the *postgres-json-schema*[5] extension. In

[5]`https://github.com/gavinwahl/postgres-json-schema`

MongoDB, this is a built-in feature that can be simply enabled with the operator `$jsonSchema`, which is provided at creation time of the collection. In tuples, all data types must always be declared, leading to constant insertion time. Oppositely, when inserting JSON, time increases with data types declaration, confirming the consequent overhead. Checking concrete ICs on top of data types, substantially increases the overhead. Both systems confirm trends, the only difference being that built-in mechanism of MongoDB being faster.

4.3 Structure complexity

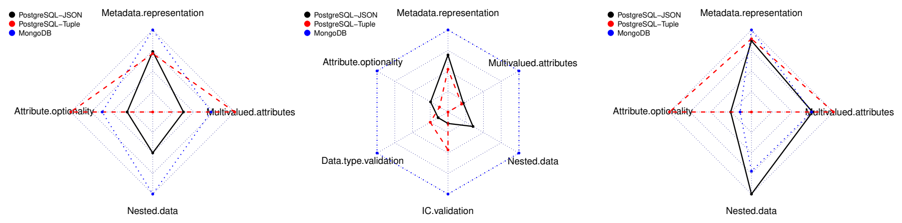
Finally, we analyse the impact of breaking first normal form by either nesting documents (Section 3.3.1) or storing multi-valued attributes (Section 3.3.2). Notice that only the latter is available in relational implementations.

Nested structures. The storage size of nesting one attribute increases the document size with the increasing number of levels. MongoDB slightly increases the physical storage when the number of levels increases, even with constant document size. The integer encoding difference (64-bits vs. 32-bits) explains this opposite behavior. The insertion time follows the same trend of the storage size. We noticed an extra overhead in MongoDB beyond that of purely I/O. PostgreSQL performs better than MongoDB (despite having higher I/O), and MongoDB have a clear upward trend with the increasing number nesting levels as opposed to constant runtime in PostgreSQL confirms the overhead nesting generates in MongoDB.

Multi-valued attributes. Regarding the storage of multi-valued attributes (Section 3.3.2), we generated documents with the number of values per attribute ranging from 2 to 64 for the different options. For tuples, we used either PostgreSQL native array storage or separate attributes for each value , and similarly for JSON either as an array in the document, or as separate attributes. Regarding storage size, both systems take more space for JSON than tuples, because of the saving of tuples on metadata replication. While in tuples both options use the same space, in JSON arrays are clearly more efficient, since separate attributes require more characters (the same behavior is confirmed in MongoDB, but mitigated by its smaller encoding of integers). despite insertion time in JSON is dominated by I/O, in tuples inserting to an array is faster than inserting multiple attributes, due to the overhead of parsing and validating independent attributes in front of one single array. Nevertheless, the extra processing at insertion time pays off at query time, where processing the independent attributes is faster than digging inside the array. For JSON, we appreciate the same benefit of querying independent attributes in PostgreSQL, but surprisingly the opposite behavior in MongoDB, where processing the array is systematically faster. When summin indivudual attributes, MongoDB has a built-in function that sums the content of the array, which is more efficient, and on the contrary, PostgreSQL needs to unwind the array in order to calculate the sum, which is more expensive.

5 Discussion

Fig. 6 summarizes all results with regard to storage space, load time, and query time. For this, we calculated the average of all measurements per representational difference for each of the three options (i.e., Tuples, and JSON in both systems). Since data follows different patterns in each case, we separately min-normalize per case (e.g., divide the minimum of the three averages for nested data by the average for Tuples) and plot them all in the corresponding radar chart. This means values further away from the center of the radar are better than the ones closer, and the bigger the area of the polygon, the better the system performs.



(a) Storage (b) Insertion time (c) Query time

Fig. 6: Multidimensional view of experimental results

According to Fig. 6a, storing tuples takes the least amount of space in all cases except metadata representation. On interpreting this, we acknowledge the impact of the ratio between metadata and data, which is fixed to be relatively high in all experiments. Thus, attribute names should always be encoded in JSON to shorten them as much as possible and improve that ratio. Obviously, this is more relevant, for example, if values are numeric than if they are strings (the former requiring less space, in general). Within JSON, PostgreSQL storage size is much larger than MongoDB in all the cases, due to the different encoding of integers (64-bits vs. 32-bits).

According to Fig. 6b, it is clear looking at PostgreSQL that loading JSON is faster than tuples, except for data type and integrity constraint validations. However, it is important to note that the validation of JSON was carried out through a third-party plugin, which definitely impacts the results. MongoDB being a native document store, has a clear advantage over PostgreSQL JSON storage in loading data (at the end of the day, JSON is stored as a column in a PostgreSQL table), beating even tuple storage in the validation dimensions. This, however, can come not only from using JSON format but from other DBMS characteristics (e.g., lack of ACID transactional support).

Finally, Fig. 6c depicts that tuples, in general, perform better in queries. This is so because they use less space, in general, and benefit from validation at insertion time. Thus, we can see that when the space-saving is lost depending on the data-metadata ratio, so the benefit is mostly lost at query time, as well. Nonetheless, JSON representation is at a disadvantage, as each of the documents needs to be parsed and processed on demand. Consequently, we should consider the trade-off between the pressure of fast ingestion and the long term benefit of recurring queries. It is also interesting to see that even though the storage size of JSON is larger in PostgreSQL, this is still faster than MongoDB. We believe this fact results from the differences in how query engines handle the calculations.

PostgreSQL benefits here from the well-optimized aggregation operations in the relational engine, which data stored in JSON format also have access to.

6 Conclusions and future work

In this paper, we studied the impact of physical design choices for NOSQL databases according to six different characteristics. We conclude that there is no *ace of spades*, when designing JSON documents. However, we identified a crucial trade-off between insertion and query performance. Nowadays, organizations are shifting their data repositories to flexible representations following a *schema-on-read* approach, but we have empirically shown that such an approach might have several shortcomings in front of query-intensive workloads. As future work, we aim to extend our experiments taking into account more features from the DBMS in use. This involves considering caching mechanisms or indexing structures.

References

1. S. Abiteboul. Querying Semi-Structured Data. In *ICDT*, 1997.
2. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.
3. S. Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley& Sons, 2003.
4. P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone. Data modeling in the NoSQL world. *Comput. Stand. Interfaces*, 67, 2020.
5. A. Badia and D. Lemire. A call to arms: revisiting database design. *SIGMOD Rec.*, 40(3):61–69, 2011.
6. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM*, 13(6):377–387, 1970.
7. A. de la Vega, D. García-Saiz, C. Blanco, M. E. Zorrilla, and P. Sánchez. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models. *Future Gener. Comput. Syst.*, 105:455–474, 2020.
8. A. Hernández, F. Santiago, E. Calvo, G. Herzig, S. A. Ostapowicz, M. Melli, and J. D. Fernández. Performance Benchmark PostgreSQL/MongoDB (Tech. R.). 2019.
9. V. Herrero, A. Abelló, and O. Romero. NOSQL design for analytical workloads: Variability matters. In *ER*, 2016.
10. M. Hewasinghage, A. Abelló, J. Varga, and E. Zimányi. DocDesign: Cost-Based Database Design for Document Stores. In *SSDBM*, 2020.
11. A. Kanade, A. Gopal, and S. Kanade. A study of normalization and embedding in MongoDB. In *IACC*, 2014.
12. C. Mohan. History repeats itself: sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT*, 2013.
13. S. Scherzinger and S. Sidortschuck. An Empirical Study on the Design and Evolution of NoSQL Database Schemas. *CoRR*, abs/2003.00054, 2020.
14. P. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.
15. C. Truica, F. Radulescu, A. Boicea, and I. Bucur. Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database. In *CSCS*, 2015.