Knowledge and Information Systems (2023) 65:3045–3078
https://doi.org/10.1007/s10115-023-01828-3

REGULAR PAPER

Automated database design for document stores with multicriteria optimization

Moditha Hewasinghage[1] · Sergi Nadal[1] · Alberto Abelló[1] · Esteban Zimányi[2]

Received: 28 September 2021 / Revised: 31 December 2022 / Accepted: 6 January 2023 /
Published online: 11 March 2023

Abstract
Document stores have gained popularity among NoSQL systems mainly due to the semi-structured data storage structure and the enhanced query capabilities. The database design in document stores expands beyond the first normal form by encouraging de-normalization through nesting. This hinders the process, as the number of alternatives grows exponentially with multiple choices in nesting (including different levels) and referencing (including the direction of the reference). Due to this complexity, document store data design is mostly carried out in trial-and-error or ad-hoc rule-based approaches. However, the choices affect multiple, often conflicting, aspects such as query performance, storage space, and complexity of the documents. To overcome these issues, in this paper, we apply multicriteria optimization. Our approach is driven by a query workload and a set of optimization objectives. First, we formalize a canonical model to represent alternative designs and introduce an algebra of transformations that can systematically modify a design. Then, using these transformations, we implement a local search algorithm driven by a loss function that can propose near-optimal designs with high probability. Finally, we compare our prototype against an existing document store data design solution purely driven by query cost, where our proposed designs have better performance and are more compact with less redundancy.

Keywords Document store · Database design · Optimization

✉ Moditha Hewasinghage
moditha@essi.upc.edu

Sergi Nadal
snadal@essi.upc.edu

Alberto Abelló
aabello@essi.upc.edu

Esteban Zimányi
ezimanyi@ulb.ac.be

[1] Universitat Politècnica de Catalunya, BarcelonaTech, Barcelona, Spain

[2] Université libre de Bruxelles, 1050 Bruxelles, Belgium

1 Introduction

In the last couple of decades, the data storage paradigm has shifted from traditional relational database management systems (RDBMSs) toward more flexible NoSQL engines [1]. Among these, the popularity of document stores has prevailed due to the adoption of semi-structured data models, which avoids the impedance mismatch between the data storage and applications. Document stores allow users to focus on rapid application development with a data-first approach instead of the traditional schema-first of RDBMSs. This has caused an increase in their popularity, especially in the startup ecosystem, where the goal is to rapidly deliver products into the competitive market [2]. As a consequence, database design (i.e., the design of database structures and their relationships) has been overlooked and not performed in a principled manner. However, it has been shown that the choice of database design plays a critical role in performance [3]. Database design for document stores is, in general, mostly carried out in a trial-and-error or ad-hoc rule-based manner. For instance, MongoDB, the leading document store, provides a set of design patterns that define certain guidelines on how to structure documents.[1] Nevertheless, even with these guidelines, it is still common to make bad design decisions, and issues tend to arise in the long run when the amount of data has grown considerably, and changing them incur additional cost, time, and money.

Let us consider an exemplary scenario of product reviews composed of the entities *Comments* and *Products*, with a one-to-many relationship from the latter to the former, as well as an equiprobable hypothetical workload defined as follows: *(a)* given a Comment ID, find its text; *(b)* given a Product ID, find its name; *(c)* given a Comment ID, find the Product name; and *(d)* given a Product ID, find all of its Comments. To illustrate the complexity of manually determining the optimal design even in such an oversimplistic scenario, we conducted a questionnaire to database experts on an equivalent setting.[2] 63% of the participants managed to identify only three potential designs for a document store, many of them overlooking the redundant nesting and referencing options. After comparing the results against an actual experimental setup performance on MongoDB, we concluded that only 9% of the participants managed to find the optimal design, while 40% guessed the fourth-best design as the optimal one (in terms of query performance). This evidences that the current way of database design does not yield the expected results, even for very limited scenarios like this. Indeed, real-world scenarios are far more complex involving multiple entities and relationships.

If we assume that all attributes of an entity are kept together within the same document, we are still left with the decision on where the relationships must be stored in the final design. Thus, database designs can be enumerated based on the alternatives to store the relationships, which depend on three independent choices: Direction, Representing, and Structuring, as shown in Fig. 1 together with two examples. **Direction** determines which entity keeps the information about the relationship. It can be any of the two entities or both. **Representation** affects how this relationship is stored by either keeping a reference or embedding the object. Finally, **Structuring** determines how we structure the relationship, either as a nested list or flattened (in the case of one-to-many relationships). For example, if we decide to keep the references to the comments in the product, they can be stored as a list of references (*comment:...*) or in a flattened manner (*comment_1:.., comment_2:..*). Hence, we end up with 24 possible designs for our running example. Depending on end-user preference, each one of these designs has the potential to become the optimal design choice. This trade-off between alternatives makes the process of finding the optimal design a complex one.

[1] https://www.mongodb.com/blog/post/building-with-patterns-a-summary.

[2] https://moditha.typeform.com/to/NRTjEm.

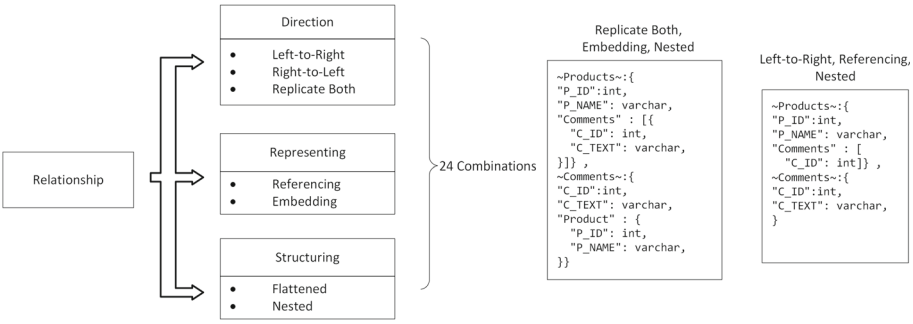


Fig. 1 Relationship storage choices for database design

As a matter of fact, the number of relationships r determines the number of candidate designs, which is exponential (24^r), as the storage option of each relationship is independent of others. Note, however, that here we did not consider allowing heterogeneous collections/lists, which is possible in the context of schemaless databases, leading to a complexity increase. For example, collections at the top level could potentially contain different kinds of documents. In our running example, the product and comment documents could be stored in a single heterogeneous collection mixing both. Precisely, for a design with c top-level collections, the total number of combinations will be $\sum_{i=1}^{c} \left\{ {c \atop i} \right\}$, where $\left\{ {n \atop k} \right\}$ is the Stirling number of the second kind, corresponding to the number of ways to partition n distinct elements into k non-empty subsets [4]. Overall, such exponential growth makes it impossible to enumerate and evaluate all candidate designs.

To overcome such limitations, there exist several solutions in the literature mainly relying on the query workload to propose a database design, such as DBSR [5], NoSE [6], and Mortadelo [7]. However, query performance is not the only factor affected by design choices. For instance, having redundant collections to support different queries will increase query performance but require larger storage capacity. Having complex document structures with multiple branching and nesting levels hinders the readability of documents. Similarly, having heterogeneous collections requires additional documentation to map where each piece of information is stored. In our previous work, we enabled users to evaluate alternative designs of their choice [8]. However, this approach is painstaking for them and highly dependent on their competencies. Therefore, we believe that it is essential to find an optimal design that considers a variety of the user's preferences automatically. To that end, we propose a novel automated database design method for document stores that considers all such factors. This is achieved by the adoption of well-known multicriteria optimization techniques, proven to efficiently tackle multiple and conflicting objectives [9]. Thus, we generate near-optimal database designs that balance the end-user's preferences regarding four quantifiable objectives: storage size, query performance, degree of heterogeneity and the average depth of documents. This work has been demonstrated as a tool DocDesign 2.0 [10]. To show the effectiveness of our method, we compare it against an existing document store schema generation tool in terms of the quality of the generated design and its performance. Our experimental results show that we manage to present a design with less redundancy and offer better performance with the flexibility of catering to the end-user preference. We can also scale up when the number of entities and relationships grows.

Contributions. The main contributions of our work are as follows:

- We propose a novel loss function for multicriteria selection of the optimal database design for document stores.
- We devise an algebra of transformations that allow to systematically modify a database design while retaining all the information (i.e., no attributes or entities are lost).
- We present an implementation of a local search algorithm that, driven by the loss function, proposes with high probability near-optimal designs.
- We assess our method and prototype to show the scalability as well as the performance gain of our solution against competitors by using the RUBiS benchmark [11].

This paper is organized as follows. First, in Sect. 2, we discuss related work. We provide an overview of our approach in Sect. 3. In Sect. 4, we introduce the canonical model used to represent the search space of alternative designs. Next, in Sect. 5, we formalize the design process over the canonical model with random state generation and design transformations. Then, we validate our data design approach with extensive experiments in Sect. 6. Finally, we conclude our work in Sect. 7.

2 Related work

JSON has gained popularity in recent years as an alternative storage format to XML. Although JSON is semi-structured, a schema can be defined [12], and thus some approaches aim to extract such schema representation from heterogeneous JSON documents [13]. When it comes to data design for document stores, these mainly rely on finding a solution that balances the two extremes (i.e., normalized and embedded models) [14]. It has been shown that the design decisions for JSON storage are not trivial and can affect the performance as storage requirements that also depends on the choice of the storage system [15]. Moreover, some authors [16] discuss data quality aspects that could arise in JSON stores and how to measure them, which can help to evaluate the potential document designs.

It is well-known that design methods for NoSQL data stores must consider both relational and co-relational perspectives at once [17]. Nevertheless, although data modeling has played a significant role in RDBMS, little work has been carried out on data design methods for document stores. Different approaches have been employed to tackle such problems, including some for analytical workloads [18], as well as cost-based schema design evaluation [8] based on a hypergraph data model [19]. NoAM [3] uses three constructs (collection, block, and key) to introduce an abstract data model that can be mapped into heterogeneous NoSQL stores based on entity aggregates. Alternatively, the SOS platform [20] introduces three design constructs (attributes, structs, and sets), which are capable of representing relational, key-value, document, and column-family stores. Indeed, our approach builds on a hypergraph-based formalization of the SOS constructs, which we proposed in a recent work [19]. This formalization enables the generation of native queries over the heterogeneous stores to manage the metadata of polyglot systems, thus representing the basis for defining design transformations.

Additionally, several works have been carried out specifically on automated schema design for NoSQL stores. We followed the principles and guidelines of Systematic Literature Reviews (SLR) as established by Kitchenham and Charters [21]. We used the search terms *NoSQL design*, *document store schema*, *NoSQL schema* as the search terms to select the initial set of articles from DBLP [3] and used a snowballing approach to track down references related on automated schema design on document stores and NoSQL systems in general. However, there are only a few schema generation/suggestion tools available for

[3] https://dblp.org.

Table 1 Overview of existing tools for schema generation

System	Data store	Workload	Optimization	Exploration
Chebotko et al. [24]	Col	Read	Rule-based	Low
Mortadelo [7]	Col/Doc	Read	Query cost/metamodel	Low
De Lima et al. [23]	Doc	Read	Query cost/rule-based	Low
NoSE [6]	Col	Read	Query cost	Low
DBSR [5]	Doc	Read	Query cost	Low
DocDesign 2.0	Doc	Read	Multicriteria	High

NoSQL systems. DBSR [5] is a database schema recommender for document stores, which uses a search-based approach to navigate the data design space similar to the one of this paper. However, DBSR only considers the query workload in generating potential designs. Moreover, DBSR only considers nesting as opposed to referencing, avoiding joins altogether. However, in our work [8] we have encountered instances where external joins perform better than certain embedded approaches. Thus, DBSR can omit some optimal designs in its process. DBSR compares itself to other existing approaches that we found relevant for our work; hence, we have compared our approach against it, because of being it is the latest research carried out in automated schema design and specific to document stores and superior to previous ones. There are several similarities between DocDesign 2.0 and DBSR they are both based on read-only workloads, and allow data duplication, nested structures, secondary index usage, and query plan estimation on document stores. However, the cost model [22] used by DocDesign 2.0 is more mature and robust to the underlying document store implementation as opposed to the simple access pattern-based one used by DBSR. Moreover, DocDesign 2.0 uses a multi-criteria-based approach with more fine tune capabilities to the user, compared to only tuning the number of final collections in DBSR resulting in a superior output.

NoSE [6] uses a cost-based schema design approach specific for column stores. In Mortadelo [7], a model-driven data design based on a generic data model is used to generate optimized data store schemas and queries for document and column stores. Another approach [23] generates a physical schema from a logical one for document stores based on the workload by optimizing the query access patterns. Unlike rule-based design generation in these approaches that could omit certain designs, our work opens the door to potentially generate all possible combinations.

Table 1 summarizes the existing schema generation tools for NoSQL systems. Next to the name of each system, we find in the first column the kind of system it is, because both document and column stores are semi-structured and benefit from the same kind of design improvements. Thus, the approaches taken to optimize their design are comparable, albeit document stores having complex query capabilities. Irrespectively of the kind of system, all of the observed approaches utilize random read workloads. Nevertheless, all other approaches except ours use only the query cost as the goal of optimization, while DocDesign 2.0 considers multiple criteria for greater flexibility in the design. Moreover, DocDesign 2.0 systematically explores the search space (possible designs) thanks to our algebra of transformations, as compared to the others which are only guided by the query workload or heuristic rules.

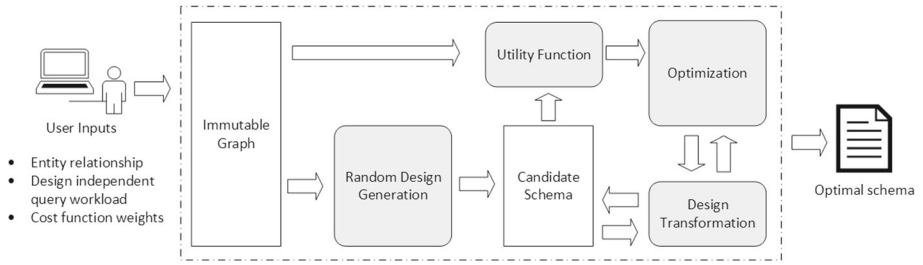
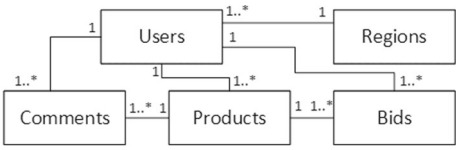


Fig. 2 High-level overview of our approach

Fig. 3 ER diagram for RUBiS framework



3 Overview

In this section, we provide a high-level overview of the components of our approach. In order to yield the most suitable database design for a given set of contradicting objectives, we adopt multicriteria optimization techniques. These have shown to be effective in obtaining near-optimal solutions out of a large search space in the presence of conflicting objectives [25]. In these scenarios, one can only aim to obtain a Pareto-optimal solution (a solution that, in the presence of multiple objectives, cannot improve one objective without worsening another). An overview of our approach is shown in Fig. 2; next in the following subsections, we briefly describe each of its components: namely user inputs, the design process, loss function, and the search algorithm.

Let us take the RUBiS benchmark [11] as a running example. As depicted in Fig. 3, RUBiS implements an online auction system (we also take the same 11 queries used by the DBSR framework as our workload).[4] If one were to design a traditional RDBMS data storage for this case, it would be natural to apply normalization and deploy a database schema in either third-normal form (3NF) or Boyce–Codd normal form (BCNF). It is well known from relational theory that such a design avoids update anomalies and would still efficiently execute many queries. However, this penalty is not acceptable for document stores, and they encourage denormalization and promote nesting to keep related pieces of information together, avoiding expensive joins even if increasing redundancy. It is thus unclear what would be the best database design, whether it could be a fully denormalized collection with `Regions` as the top-level document, a normalized approach similar to 3NF, or an in-between solution. With reference to our estimate in the introduction, there are 24^6 possible alternative designs to choose from.

3.1 User inputs

The end-user must provide three inputs to initiate the optimization process: the Entity-Relationship (ER) diagram (enriched with some physical information like attribute sizes

[4] https://github.com/vreniers/DBSR.

and cardinalities), a query workload (i.e., a set of design-independent queries together with their frequencies), and the weights of the four cost functions.

Entity-Relationship describes the domain in terms of a graph and determines the complexity of the optimization problem. The user has to identify the entities, the attributes of each entity, the average size of the attributes, the number of instances of the entities, and their relationships, including the multiplicities. As previously mentioned, we represent this information using a hypergraph-based canonical model composed of *Atoms* and *Relationships* [19]. In our running example, these are the entities, relationships, and multiplicities shown in Fig. 3. By definition, this graph is considered immutable, and we carry out the database design on top of it by building hyperedges. We discuss the canonical model and the design process in detail in Sect. 4.

Query workload determines the user's query requirements on the underlying data. Since the input ER does not contain design information, the queries are also represented in a design-independent manner. Thus, a query is a set of interconnected *Atoms* with a specific one representing a selection criterion (as defined in the cost model [22]). Our approach works with a constant workload, including the frequency of each of the queries. In our running example, the workload would be the 11 queries in RUBiS together with their frequencies.

Cost function weights determine how the final design performs in four criteria, namely: query performance, storage size, degree of heterogeneity within a collection, and depth of the documents. Each of these has a cost function associated which will be weighted according to the user's needs to compose the loss function to be optimized.

3.2 Design processes

Once the user has provided the ER diagram, it is stored in an immutable graph where the entities and attributes are represented as vertices (*Atoms*) and relationships as edges [19]. Precisely, we perform all design operations on top of this immutable graph generating candidate designs. Two processes generate these candidate designs, namely: initial **random design generation** (through those relationship design choices shown in Fig. 1) and **design transformation** of an existing design (through the methods associated with the different classes in the canonical model). For instance, the random design generator could generate a design with five separate collections for each entity with their respective references (e.g., the user references the region). Then, through design transformations, we can generate an alternative design from the existing one by embedding the region inside the user. We introduce the formal canonical model, algorithms to generate random designs, and design transformations in Sect. 4.

3.3 Loss function

We introduce four components of the loss function to be measured and optimized in DocDesign 2.0 : query cost, storage cost, degree of heterogeneity, and the average depth of the documents (as a measure of their complexity). We chose these loss functions as a representative of different cost functions and not intended as an ultimate combination that would provide the optimal database design. These are defined as follows:

Query cost (CF_Q) is the weighted average of the relative query performance values estimated from the schema information using a cost model for document stores [22]. This cost model is

configurable according to the storage and the execution model of different document stores. Thus, it is possible to adapt DocDesign 2.0 to alternate document store implementations. The cost model firstly takes the query workload as an input, calculates the distribution of the cache under a workload, and estimates a relative cost of accessing each of the collections and indexes. Then, the total relative cost of each of the queries (Q_q) that depends on the access patterns of the storage structures is also calculated. Thus, we can sum up the total query cost as $CF_Q = \sum_{k=1}^{N_q} f^k \cdot Q_q^k$ where N_q is the number of queries in the workload and f^k is the frequency of the Query.

Storage size (CF_S). Is the total storage size required by the collections and indexes, calculated using schema information in the canonical model [19]. We define the size of a collection and an index as S_C and S_I, respectively. Thus, the total storage size $CF_S = \sum_{k=1}^{N_c} S_C^k + \sum_{k=1}^{N_i} S_I^k$ where N_c and N_i are the total number of collections and indexes in the design.

Degree of heterogeneity (CF_H) is the number of distinct types of documents in a collection/list. We use the weighted average over all the collections and lists of the schema. Each value is given a weight depending on which level the list/collection lies in the document. The higher the level, the higher the assigned weight, thus penalizing heterogeneities at higher levels of the document structure. If there are n heterogeneous collection/list at a given level lv, the degree of heterogeneity value $H_{lv} = \frac{n}{lv+1}$. Assuming there are N_h number of collections/lists in the design, the average degree of heterogeneity $CF_H = \frac{\sum_{k=1}^{N_h} H_{lv}^k}{N_h}$.

Depth of the documents (CF_D) is the average depth of the documents of the design. We can assume that each document at the top level of a collection has branches reaching from the root to the leaf level with a length ln, and there is N_b number of branches in all the documents in the top level. Thus, we define the average depth of the documents of the design $CF_D = \frac{\sum_{k=1}^{N_b} ln^k}{N_b}$.

3.4 Search algorithm

Local search algorithms consist of the systematic modification of a given state, utilizing action functions, in order to derive an improved state. The intricacy of these algorithms consists of their parametrization, which is, at the same time, their key performance aspect. Due to the genericity of different use cases our method can tackle, we decided to choose *hill-climbing*, a nonparametrized search algorithm that can be seen as a local search, always following the path that yields lower loss values. Nevertheless, the cost functions we use are highly variable and non-monotonic, which can cause hill-climbing to provide different outputs depending on the initial state. To overcome this problem, we adopt a variant named *shotgun hill-climbing*, which consists of a hill-climbing with restarts using random initial states.

Loss function. Guiding the local search algorithm requires the definition of a loss function, taking into account the end-user preferences. Here, this is a function to be minimized. Hence, the end-user can assign weights to each cost function according to their importance in the use case. Then, for a given design C, we define the loss as the normalized weighted sum of each cost function $l(C) = \sum_{k=1}^{n} w_k \frac{CF_k(C) - CF_k^{\min}}{CF_k^{\max} - CF_k^{\min}}$. The expression considers the weight w_k of each cost function, which is used on the transformed loss function for C. This is a normalized value that considers the *utopia* (i.e., the expected minimal) and the maximal

Algorithm 1 Shotgun Hill-Climbing

Input: [Initial design, number of non-improving iterations] D, N:
Output: [Solution design] $solution$

```
1: solution ← null; i ← N
2: while i > 0 do
3:    B ← randomInitialState(D); finished ← false
4:    while !finished do
5:       neighbors ← applyAllPossibleTransformations(B)
6:       B′ ← stateWithSmallestLoss(neighbors)
7:       if l(B′) < l(B) then
8:          B ← B′
9:       else
10:         finished ← true
11:      end if
12:   end while
13:   if l(B) < l(solution) then
14:      solution ← B
15:      i ← N
16:   end if
17:   −−i
18: end while
```

design costs, yielding values between zero and one, depending on the accuracy of both $CF_i^{\min}$ and $CF_i^{\max}$, which rely on estimations.

Shotgun hill-climbing. Algorithm 1 depicts an overview of shotgun hill-climbing. The method generates random designs and systematically improves them by applying at each step all available transformations keeping the one that yields the minimum loss value. Note, however, that this approach is highly susceptible to fall in local minimums. To overcome this issue, we repeat the process a certain number of iterations and keep the one with the minimum loss, overall. Once such a solution is not improved for a certain number of iterations denoted by N (i.e., we do not find any new state with a smaller loss), it is highly probable that we have found the optimal design.

Thus, DocDesign 2.0 will provide a pareto-optimal design depending on the importance given to the each of the loss functions and the number of non-improving iterations given by the user. For instance, retaking the running example, if the user had specified to optimize on the storage space, the solution would be individual collections with references with minimal or no nested documents (Listing 1). Or, if the query performance is the only important factor, it is likely to generate collections that can answer the queries without joins increasing the redundancy (Listing 2).

Listing 1 Optimized for storage

```
"REGION": {
   "R_ID": int(4),
   "R_NAME": varchar(10)
}
"USER": {
   "U_ID": int(4),
   "U_F_NAME": varchar(20),
   "R_ID": int(4),
}
"PRODUCT": {
   "P_ID": int(4),
   "P_TITLE": varchar(10),
   "C_ID": int(4),
   "B_ID": int(4),
   "U_ID": int(4)
}
"COMMENT": {
     "C_ID": int(4),
     "C_TITLE": varchar(20),
     "U_ID": int(4),
     "P_ID": int(4)
}
"BID": {
     "B_ID": int(4),
     "B_PRICE": int(6),
     "U_ID": int(4)
 }
```

Listing 2 Optimized for queries

```
"USER-BIDS": {
   "U_ID": int(4),...
   "REGION": {
     "R_ID": int(4),...
   },
   "BIDS": [{
     "B_ID": int(4),...
   }]
}
"USER-COMMENTS": {
   "U_ID": int(4),...
   "R_ID": int(4),
   "REGION": {
     "R_ID": int(4),...
   },
   "COMMENTS": [{
     "C_ID": int(4),...
     "U_ID": int(4)
   }]
}
"PRODUCT-COMMENT": {
   "P_ID": int(4),...
   "COMMENTS": [{
     "C_ID": int(4),...
     "U_ID": int(4)
   }],
   "B_ID": int(4),
   "U_ID": int(4)
}
"BID-USER": {
     "B_ID": int(4),...,
     "U_ID": int(4),...
 }
```

4 Canonical model

In this section, we present the canonical data model we use to represent database designs for document stores. We extend our previous work [19], where we presented a hypergraph-based canonical data model for polyglot systems. A hypergraph is a generalization of a graph in which an edge can join more than one vertex. Here, we extend it with additional artifacts and class methods specific to document stores. To that end, we distinguish three levels of detail from most abstract to most specific (immutable, storage agnostic, and document store-specific constructs). Each subsection corresponds to the three abstraction levels. From now on, we will use the conventions listed in Table 2. The subscripts R, H, $Struct$, Set, Doc, Top, $List$, and Col of E denotes the type of the edges such as *relationship*, *hyperedge*, *struct*, *set*, *document*, *top level document*, *list*, and *collection*, respectively. The superscript is used to denote the type of the data store. In this paper this is always Doc for document stores.

4.1 Immutable graph

Figure 4 depicts the main constructs of our canonical data model with the three levels of abstraction highlighted. The ER diagram provided by the user (e.g., Comments and Products) is considered immutable. This immutable information is a simple graph consisting of *Atoms* (A) and *Relationships* (E_R). An *Atom* is either an attribute A_A or a class representative A_C (e.g., comment text and comment ID, respectively). This immutable graph can only represent binary relationships of an ER diagram. However, the representation of ternary relationships is also possible by reification of the relationship and transforming into binary relationships.

Over the immutable metadata, we define the different design artifacts depending on the model being used (e.g., documents in our case). Therefore, we define design operations for those artifacts at two specialization levels, each containing different class methods on the

Table 2 Variables used in the paper

C	The catalog (the design)	E_N	Node (either atom or edge)
A	Atom	$\mathbb{A}$	Set of all atoms in C
A_C	Class atom	$O(h)$	Root atom of a *struct* h
$E_R^{x,y}$	Directed relationship between two atoms x and y	$\mathbb{E}_R$	Set of all relationships in C
E_H	Hyperedge	E_H^+	Transitive closure of E_H
E_{Struct}	*Struct*	$\mathbb{E}_{Struct}$	Set of all the E_{Struct}s in C
E_{Set}	*Set*	$\mathbb{E}_{Set}$	Set of all the E_{Set}s in C
E_{Doc}^{Doc}	Hyperedge representing a document of a document store	E_{Top}^{Doc}	E_{Doc}^{Doc} representing top level document of a collection of a document store
E_{List}^{Doc}	Hyperedge representing a list in a document store	E_{Col}^{Doc}	E_{List}^{Doc} representing a top level list (a.k.a collection)
$E_R^{A_C,A_C}$	Relationship between two class atoms	E	Generic superclass for edges

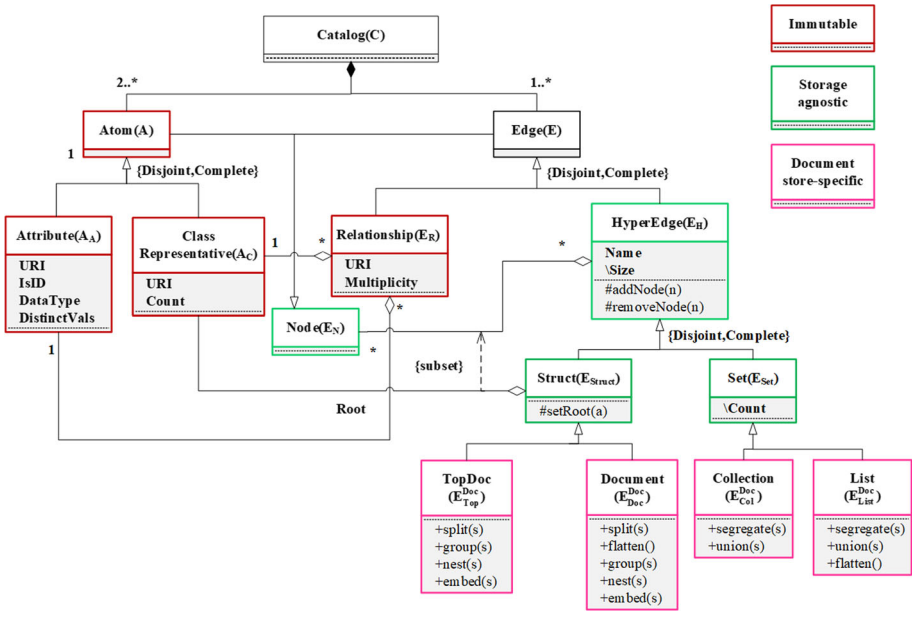


Fig. 4 Class diagram of the canonical model

canonical model, getting more concrete as the specialization progresses. Operations at an abstraction level have access and use the operations at more generic levels above.

4.2 Storage-agnostic constructs

The main design construct of our model is a *Hyperedge* (E_H). By definition, a hyperedge is an edge that connects more than one vertex in a graph. We use the concept of generalized

Table 3 Hypergraph methods

Method	$\langle\langle preconditions \rangle\rangle$	$\langle\langle postconditions \rangle\rangle$
$Hyperedge(C, nodes : Set\ of\ E_N)$		• $nodes \subseteq self$
		• $self \in C$
$\backsim Hyperedge()$		• $self \notin self.C$
		• $self.children@pre \subseteq self.parent$
$E_H.addNode(n : E_N)$	• $self \notin n^+$	• $n \in self$
$E_H.removeNode(n : E_N)$	• $n \in self$	• $n \notin self$

hypergraph where a hyperedge can also contain other hyperedges. *Hyperedges* (E_H) in our canonical model consists of a set of *Nodes* (E_N), which can be either *Atoms* (A), *Relationships* (E_R) or other *Hyperedges* (E_H).

In Table 3, we identify methods involving *Hyperedges*. These hypergraph operations are independent of the design constructs and the concrete data store, and we show them directly through pre- and postconditions. Firstly, creating a *Hyperedge* defines a set of nodes in the catalog. On destroying it, all nodes inside are absorbed by its parent to ensure the validity of the catalog as per Definition 3. On adding a new node E_N to a *Hyperedge* E_H, it must happen that the *Hyperedge* E_H does not transitively contain itself to avoid cyclic references. Here, $self$ and $@pre$ refer to the instance of the E_H and value of the corresponding variable before the operation, respectively.

Now, we introduce two specialized E_Hs that are generic to any database system. E_{Struct} represents the structure of the stored elements in the database (i.e., a row in an RDBMS or a document in a document store) with a specific A_C identified as the *root* (primary attribute of the E_{Struct}). E_{Set} represents the collections of the database (table in an RDBMS, list or collection in a document store), which can contain different kinds of structures (in the case of document stores). We use the hypergraph operations to manipulate specialized hyperedges E_{Struct} and E_{Set}, defined at Definitions 1 and 2.

Definition 1 A *Struct* is a subclass of *Hyperedge* with a prominent node inside (called *root*, noted $O(self)$, which is an A_C), so that:

(a) All the *Atoms* in a *Struct* must have a single path of relationships to its *root* which is also part of the *Struct*.
$\forall a \in (self \cap \mathbb{A}) - O(self) : \exists!\{E_R^{O(self),x_1}, \ldots, E_R^{x_n,a}\} \subseteq self$

(b) All the *roots* of the nested *Structs* inside a parent *Struct* must have a single path of relationships from the *root* of the parent, and this path must be inside the parent.
$\forall s \in (self \cap \mathbb{E}_{Struct}) : O(self) = O(s) \vee \exists!\{E_R^{O(self),x_1}, \ldots, E_R^{x_n,O(s)}\} \subseteq self$

(c) All the *Sets* inside a parent *Struct* must contain a set of relationships that connect any *Atom* of the parent to the *root* of the child *Struct* or a child *Atom* of the *Set*.
$\forall s \in (self \cap \mathbb{E}_{Set}), \forall t \in (s \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : \exists!\{E_R^{y,x_1}, \ldots, E_R^{x_n,z}\} \subseteq s \wedge y \in (self \cap \mathbb{A}) \wedge (t \in \mathbb{A}\ ?\ z = t : z = O(t))$

(d) To make sure that there are no dangling relationships inside the *Struct*, all of the relationships inside it must be involved in a path that connects either the child *Atoms* or the child *Structs* to its *root*.
$\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : (\exists y \in (self \cap \mathbb{A}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \ldots, E_R^{x_n,y}\} \subseteq self) \vee (\exists y \in (self \cap \mathbb{E}_{Struct}) : E_R^{a,b} \in \{E_R^{O(self),x_1}, \ldots, E_R^{x_n,O(y)}\} \subseteq self)$

Table 4 Struct and set methods

Method	Activity
$Struct(C, r : A_C, At : Set\ of\ A, Re :$	$super(C, \{r\} \cup At \cup Re \cup Hy)$
$Set\ of\ E_R, Hy : Set\ of\ E_H, p : E_H)$	$self.setRoot(r)$
	$p.addNode(self)$
$Set(C, Re : Set\ of\ E_R, Hy : Set\ of\ E_{Struct}, At :$	$super(C, Re \cup Hy \cup At)$
$Set\ of\ A, p : E_{Struct})$	$p.addNode(self)$

Definition 2 A *Set* is a subclass of *Hyperedge*, so that:

(a) *Sets* cannot directly exist within another *Set* (i.e., they must be contained in an intermediate *Struct*).
$\forall h \in (self \cap \mathbb{E}_H) : h \in \mathbb{E}_{Struct}$

(b) Together with invariant 1c, it guarantees that all the *Structs* inside a *Set* are connected to the parent *Struct* of the *Set* by a set of relationships in the *Set* itself. Finally, all the relationships inside the *Set* must be involved in the path that connects its child *structs* or *Atoms* to the parent *Struct* to avoid dangling relationships.
$\forall E_R^{a,b} \in (self \cap \mathbb{E}_R) : \exists A_C^{x_1} \in self.parent, \exists y \in (self \cap (\mathbb{E}_{Struct} \cup \mathbb{A})) : E_R^{a,b} \in \{E_R^{x_1,x_2}, \ldots, E_R^{x_n,z}\} \subseteq self \wedge (y \in \mathbb{A}\ ?\ z = y : z = O(y))$

Table 4 shows the methods of *Set* and *Struct* constructors. Even if for the sake of simplicity, they are not explicit there; we consider all properties in Definitions 1 and 2 are invariants for these methods and consequently guaranteed also in those at document store-specific level. Here, *super* refers to the constructor of the super class.

4.3 Document store-specific constructs

Document store-specific constructs are specializations of E_{Struct}s and E_{Set}s specific to document stores. We specifically identify the document structure at the top level as E_{Top}^{Doc} and the collection as E_{Col}^{Doc}. All other documents and nested lists are identified as E_{Doc}^{Doc} and E_{List}^{Doc}, respectively. We now use the *Struct* and *Set* constructors to define the operators considering document store-specific constraints. The constraints and mappings we consider correspond to the following grammar:

$$
\begin{aligned}
C &\Longrightarrow E_{Col}^{Doc\ +} \\
E_{Col}^{Doc} &\Longrightarrow E_{Top}^{Doc\ +} \\
E_{Top}^{Doc} &\Longrightarrow A_C(A \mid E_{List}^{Doc} \mid E_{Doc}^{Doc})^* \\
E_{List}^{Doc} &\Longrightarrow E_R{}^{+}(E_{Doc}^{Doc} \mid A)^+ \\
E_{Doc}^{Doc} &\Longrightarrow A_C(A \mid E_{List}^{Doc} \mid E_{Doc}^{Doc})^*
\end{aligned}
$$

We define the constructors of the data store-specific structures considering these production rules, as shown in Table 5.

Finally, we define a valid design using these constructs, which guarantees that we do not lose any information provided in the input ER diagram.

Table 5 Document store-specific constructor methods

Method	Activity
$Document(C, r : A_C, Re : Set\ of\ E_R, At : Set\ of\ A, Do : Set\ of\ E^{Doc}_{Doc}, Li : Set\ of\ E^{Doc}_{List}, p : (E^{Doc}_{List} \cup E^{Doc}_{Struct}))$	$super(C, r, At, Re, Do \cup Li, p)$
$TopDoc(C, r : A_C, Re : Set\ of\ E_R, At : Set\ of\ A, Do : Set\ of\ E^{Doc}_{Doc}, Li : Set\ of\ E^{Doc}_{List}, p : E^{Doc}_{Col})$	$super(C, r, At, Re, Do \cup Li, p)$
$Collection(C, Do : Set\ of\ E^{Doc}_{Top})$	$super(C, \varnothing, Do, \varnothing, \varnothing)$
$List(C, Re : Set\ of\ E_R, Do : Set\ of\ E^{Doc}_{Doc}, At : Set\ of\ A, p : E^{Doc}_{Struct})$	$super(C, Re, Do, At, p)$

Definition 3 A design D is a set of collection *Hyperedges* and is *valid* if it contains all the *Atoms* and *Relationships* in the closure of at least one of its collection *Hyperedges*. **Formally**: $\forall x \in (\mathbb{A} \cup \mathbb{E}_R) : \exists E^{Doc}_{Col} \in D \wedge x \in E^{Doc+}_{Col}$.

Generating arbitrary constructs cannot guarantee a valid design as per Definition 3. Thus, when using these document store-specific constructors, the validity must be explicitly enforced.

5 Design processes over the canonical model

Now that we formally defined our canonical model to represent document store data designs, we can use it in our shotgun hill-climbing approach introduced with Algorithm 1 to find the near-optimal design. To achieve this, we need to create a initial state with a random design (line 3) and apply transformations to generate neighboring designs (line 5). Thus, we introduce two design processes over the canonical model: random design generation and design transformation each corresponding to a subsection.

5.1 Random design generation

The key concept used in the random design generator is generating connected components (i.e., subgraphs) of the immutable graph until all the *Atoms* and *Relationships* are in one of these components. This ensures that none of the input ER diagram information is lost, adhering to a valid design. Each connected component represents then a collection in the document store schema. Algorithm 2 is responsible for generating a random design together with the aid of Algorithm 3 to make the design structure decisions. The main requirement behind these algorithms is to make the relationship storage choices randomly. For the simplicity of the algorithms, we omit the flattened representation in the random generation process. Thus, a relationship can be referred, nested, or skipped (in the case of chained relationships). In our running example, the *region* collection can have *bids* embedded or referred without storing the *user* information. However, the relationship between the *users* and *bids* must be stored in another collection (i.e., *user* collection referring/embedding *bids*) to ensure no information is lost from the original ER diagram adhering to the validity of a design.

Definition 4 Each connected component (*Comp*) is represented as a tree of *Cnode*s, each representing a relationship and its storage choice, except for the root (where the *from* and the relationship are empty) and the leaves (where children are empty). Thus, each *Cnode* of the tree contains five elements: 1. the *from* A_C, 2. the *to* A_C, 3. a relationship E_R that connects the parent and the child, 4. the representation (i.e., nest, refer, or skip) of the E_R connecting them, or an indicator of being the topmost element of the component identified as the $ROOT$, and 5. the set of child *Cnode*s. **Formally**: $Comp = Cnode\langle from, to, rel, (NEST \mid REF \mid SKIP \mid ROOT), \{Cnode\}\rangle \; s.t. : from, to \in \mathbb{A}_C \wedge rel = E_R^{from,to}$

Algorithm 2 Main Algorithm

Input: *graph G* containing *Atoms* and *Relationships*

1: $allA_Cs \leftarrow G.getA_Cs()$ ▷ *all* A_C*s in G*
2: $allE_Rs \leftarrow G.getE_R^{A_C,A_C}s()$ ▷ *all* E_R*s that connect two* A_C*s in G*
3: $E_Rs \leftarrow newList\langle E_R\rangle()$ ▷ E_R*s to be explored*
4: $comps \leftarrow List\langle Comp\rangle()$ ▷ *list of connected components*
5: **repeat**
6: **if** $E_Rs \neq \varnothing$ **then** ▷ connected E_R to an explored one
7: $next \leftarrow E_Rs.remove(RandInt(E_Rs.size))$
8: $allE_Rs.remove(next)$
9: **else** ▷ pick a new random unexplored E_R
10: $next \leftarrow allE_Rs.remove(RandInt(allE_Rs.size))$
11: **end if**
12: $[root, comps, allE_Rs, allA_Cs] \leftarrow choose(next, comps, allE_Rs, allA_Cs)$
13: $E_Rs.addAll(G.getUnusedE_R^{A_C,A_C}(next.get(root)))$ ▷ *add connected* E_R*s to explore*
14: **until** $allE_Rs = \varnothing \wedge E_Rs = \varnothing$
15: **for all** $atom \in allA_Cs$ **do** ▷ make remaining A_Cs into new *Comp*s
16: $col \leftarrow newCnode(null, atom, null, ROOT)$
17: $comps.put(col)$
18: **end for**
19: **for all** $tree \in comps$ **do** ▷ transform *Comp*s into E_Hs
20: $buildHyperedge(tree, G, allA_Cs)$
21: **end for**

Algorithm 2 keeps track of unused A_Cs and E_Rs that connect two A_Cs (lines 1 and 2) and maintains a list of E_Rs to be explored and a list of connected components (lines 3 and 4). The generation process is initialized by randomly picking one of the available relationships. This can be from the list of relationships to explore (lines 6–8), if any, or from all unexplored relationships (lines 9–10). This chosen E_R will create a new connected component or extend an existing one depending on the current components using Algorithm 3, which also returns the $root$ of this connected component (e.g., assume that it is the U_ID). Then, in line 13, we take all the unused E_Rs that connect other A_Cs to the picked $root$ (e.g., $E_R^{U\_ID,R\_ID}$, $E_R^{U\_ID,C\_ID}$) expanding the connected edges to be explored in. We continue this procedure until all the E_Rs have been used for the connected components. Then, we generate new connected components for all the remaining A_Cs that are not used in any of the existing connected components (lines 15–18). Finally, in lines 19–21, we build the E_{Col}s corresponding to the connected components in G by transforming the *Comp*s into corresponding E_Hs. Due to space limitations, we introduce this transformation in Appendix A as the procedure is purely technical.

Algorithm 3 Choose Algorithm

Input: $E_R\ rel$, $List\langle Comp\rangle\ comps$, List $allE_Rs$, List $allA_Cs$

```
1: pSkip ← 0.25
2: opChoice ← flip(LEFT | RIGHT | BOTH)
3: if opChoice = LEFT then▷ need the choice otherwise we will always grow components if its connected
4:     canExtend ← false
5:     for all tree ∈ comps do
6:         for all node ∈ tree.postOrderTraversal() do
7:             if node.to = rel(0) ∧ node.from ≠ rel(1) ∧ (∄ n ∈ node.children s.t n.rel = rel) then▷ find
       a node with rel(0) as "to" which is not connected by rel(1) and has no children or none of the children has
       used rel
8:                 node.addChild(rel(0), rel(1)), rel, flip(REF | NEST))          ▷ add new child to the tree
9:                 allA_Cs.remove(rel(0))
10:                if randomDouble() < pSkip then                                  ▷ skip with probability
11:                    node.type = SKIP
12:                    allE_Rs.add(node.rel)                                       ▷ add the relationship back to the pool
13:                end if
14:                Connectionfound ← true
15:                break                ▷ only add the node to the tree and stop the iteration within the tree
16:            end if
17:        end for
18:        if Connectionfound then                ▷ stop looking in more trees if the node is already added
19:            break
20:        end if
21:    end for
22:    if !Connectionfound then                                                    ▷ new component
23:        root ← newCnode(null, rel(0), null, ROOT)
24:        root.addChild(newCnode(rel(0), rel(1)), rel, flip(REF | NEST)))
25:        comps.put(root)
26:    end if
27: else if opChoice = RIGHT then
28:    same as above swap 0 and 1
29: else if opChoice = BOTH then
30:    do opChoice LEFT and RIGHT (nest on both ends or refer on both ends)
31: end if
32: opChoice = BOTH ? flip(LEFT | RIGHT) : opChoice
33: return opChoice, comps, allE_Rs, allA_Cs
```

Algorithm 3 is responsible for determining the direction and the representing of a particular E_R chosen by Algorithm 2. The inputs consist of a chosen E_R, the list of currently connected components, and the list of all unused E_Rs sent by Algorithm 2. We determine the direction of the relationship randomly in line 2, which determines the from A_C of the new *Cnode*. Next, we go through the list of currently connected components (line 5), doing a postorder traversal (line 6) to determine whether one of the currently connected components can be extended with the new *Cnode* as a child. This is possible only if there is a *Cnode* with to as the *from* of the new *Cnode* and *from* is not the to of the new *Cnode* and the existing *Cnode* does not use the selected E_R in any of the children. Once we find the location, we update that connected component with the new *Cnode* with a random choice of reference or nesting (line 8).

In the case of a chain of relationships between two A_C within a connected component, it is possible to skip some of them in the final document representation. For example, we can store the list of *bids* of a particular *region* without the *user*'s details even though the *user* is related to the *bid*. The skip choice enables such design decisions. We introduce a probability to skip a relationship in line 1 and change the *Cnode* type to *SKIP* and add the E_R back to

the pool of unused E_Rs (to ensure that particular relationship information is not lost) with that probability in lines 10–13. We add the new *Cnode* to only one connected component and to a single particular branch (lines 14–21). If no component can be updated, we make a new connected component with the new *Cnode* as the $ROOT$ as shown in lines 22–31. Finally, we return the parent side of the E_R that we used back to Algorithm 2, in the case of both sides, we randomly return one of the A_Cs (lines 31–33).

The above choices are carried out until all the entities and relationships belong to at least one of the connected components. Finally, each of the components is represented as a document store collection. These initial designs do not contain heterogeneous collections or lists, yet, since we initially ignore the choice of flattening and only use the nested option for structuring concerning the options in Fig. 1. This decision reduces the complexity of the random generation and the number of starting schemas. However, we introduce this through design transformations to ensure that we do not lose certain designs in the process.

Let us consider the running example of `products` and `comments` from RUBiS and also include `users` to have a complex scenario to generate a random design. Let us assume we picked $E_R^{P\_ID,C\_ID}$ as the first E_R in Algorithm 2 line 5. Next, in Algorithm 3 we got *LEFT* as the random *opChoice* in line 2. Since there are no existing *Comp*s we move to line 21 and create a new *Comp* to the *comps* list with $Cnode\langle P\_ID, null, null, ROOT\rangle$ as the root and a single child $Cnode\langle P\_ID, C\_ID, E_R^{P\_ID,C\_ID}, NEST\rangle$ if we got *NEST* option. Now, coming back to line 7 in Algorithm 2, we have $E_R^{P\_ID,U\_ID}$ in E_Rs as the only unused $E_R^{A_C,A_C}$ connected to P_ID. Here, at line 13 we pick this E_R and go back to Algorithm 3. Let us assume that we got *RIGHT* as the *opChoice*. We cannot extend the previous *Comp* that we made as it does not satisfy the extensible criteria. Thus, we create a new *Comp* with $Cnode\langle U\_ID, null, null, ROOT\rangle$ as the root and $Cnode\langle U\_ID, P\_ID, E_R^{P\_ID,U\_ID}, REF\rangle$ as its child.

Finally, similarly, if we assume the last remaining E_R between *U_ID* and *C_ID* got *BOTH* and REF, both the *Comp*s of the *product* and the *user* will be extended with $Cnode\langle C\_ID, U\_ID, E_R^{U\_ID,C\_ID}, REF\rangle$ and $Cnode\langle U\_ID, C\_ID, E_R^{U\_ID,C\_ID}, REF\rangle$, respectively. Now that we have exhausted all E_Rs, we build the E_Hs that represent the corresponding design (algorithm in A). In this case, the design is `products` embedding `comments` in one collection with `comments` having a reference to the `users` and a second collection of `users` with a reference to both `comments` and `products`.

5.2 Design transformations

In order to generate neighboring designs to a given **valid** design, we introduce now seven public methods specific for document stores at the corresponding specific design constructs. A detailed formalization of these transformation rules is available in Appendix B.

1. **Union** : Merges two sibling E_{Set}^{Doc} into one.
2. **Segregate**: Separates a E_{Struct}^{Doc} from inside a heterogeneous E_{Set}^{Doc} into a new independent E_{Set}^{Doc}.
3. **Embed**: Embeds E_{Struct}^{Doc} into another sibling E_{Struct}^{Doc} that have a path of E_Rs.
4. **Split** Separates a E_{Struct}^{Doc} into two under a given partition of its elements.
5. **Nest** Creates a new E_{Struct}^{Doc} within an existing E_{Struct}^{Doc}, given a subset of its elements.
6. **Group** Creates a new heterogeneous E_{Set}^{Doc} containing two E_{Struct}^{Doc}s.
7. **Flatten** Removes an E_H^{Doc} and let its parent absorb the content.

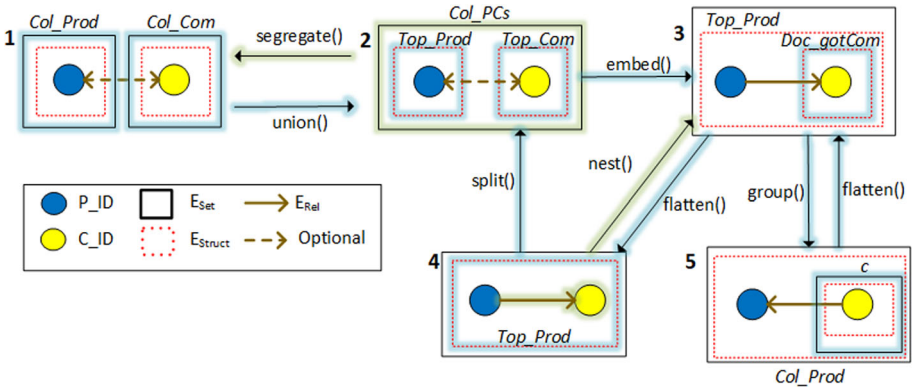


Fig. 5 Sketch of schema transformations in document stores using transformation rules

Let us retake a subset of the running example of storing only *products* and *comments* to illustrate the transformations. Figure 5 shows different designs that can be conceived and sketches the transitions between them using the transformations in Table 10. Glowing lines indicate the hyperedges that participate in the transformations that follow each design. Additionally, the hyperedge where the relationship belongs to is assumed to be that of the tail of the arrow (i.e., in `Design 3`, E_{Rel} belongs to the E^{Doc}_{Top} containing *P_ID*; in `Design 5`, it is the E^{Doc}_{Set} of *C_ID*; and in `Design 4`, that of the A_C). Notice that in `Designs 1` and `2`, there is a double-head dashed arrow, which means that, for *segregation* and *union*, its existence is optional and also can be at either one or other side. The optionality of the E_R in `Design 1` and `2` implies that the reference between the collections can reside on either side, which gives rise to four alternative schemas, namely references on both collections, one collection, or none.

From here on, to identify specific E_Rs, we introduce the corresponding As. For example, the relationship between *P_ID* and *C_ID* is written as $E_R^{P\_ID,C\_ID}$. Similarly, for the *hyperedges*, E^{Doc}_{Colpod} identify the *collection* hyperedge of `product`. Since, only A_Cs are relevant for the transformations, namely *P_ID* and *C_ID*, only these are shown to keep the figures clean. Nevertheless, we assume that the attributes of any A_C are always attached to it (e.g., *P_NAME* will always be in the same hyperedge/document as *P_ID*).

Let us assume that we start with `Design 1` where a `product` have references to the `comments` and follow the transformations as illustrated in Fig. 5. According to Table 10, in order to *union* two E^{Doc}_{Set}s, they need to share the same parent. Thus, if we call $E^{Doc}_{Col\_Prod}.union(E^{Doc}_{Col\_Com})$, firstly, the $E^{Doc}_{Col\_Com}$ is added to the source $E^{Doc}_{Col\_Prod}$), followed by the removal of the absorbed collection from the catalog. Finally, $E^{Doc}_{Col\_Com}$ is disposed, leaving its children in the new parent $E^{Doc}_{Col\_Prod}$, as represented in `Design 2`, where `comments` and `products` are in the same collection (notice that in this case only `products` references `comments`). The equivalent representation in JSON is illustrated in Fig. 6

To *embed* a E^{Doc}_{Doc} into another, they must have the same parent, and their roots must be the same, or there must be a path between them. We exemplify this by $E^{Doc}_{Top\_Prod}.embed(E^{Doc}_{Top\_Com})$, which moves $E^{Doc}_{Top\_Com}$ inside $E^{Doc}_{Top\_Prod}$ as in `Design 3`. The result is that for each *product*, there will be several `comments` in the form of a flattened list, and each document will carry the name of the relationship suffixed by a counter.

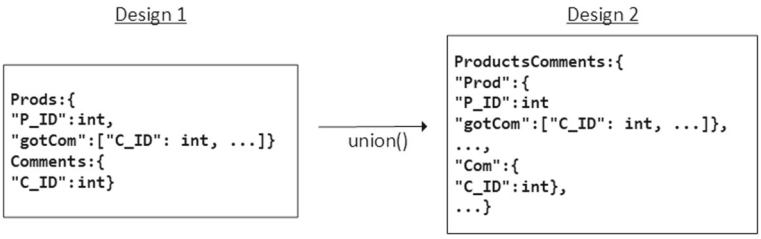


Fig. 6 Union transformation

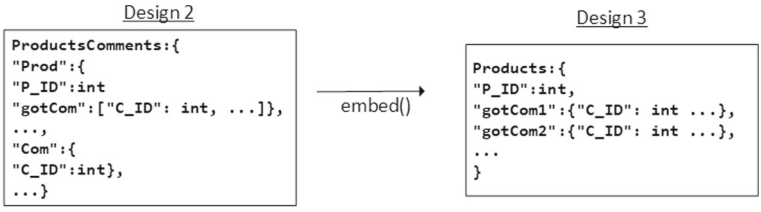


Fig. 7 Embed transformation

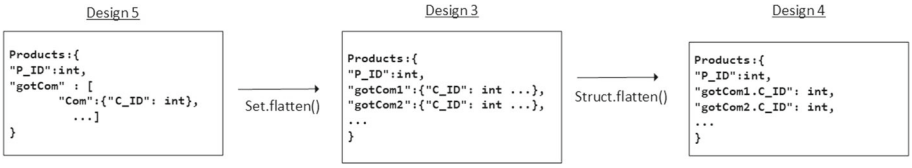


Fig. 8 Flatten transformation

Implementation-wise, we rename the embedded E^{Doc}_{Doc} with the name of the relationship followed by a counter, as shown in the JSON in RHS of Fig. 7.

In order to *flatten* an E_H, its parent must be a E^{Doc}_{Doc} (or E^{Doc}_{Top}) to ensure a correct design (i.e., sets directly inside *sets* should not be allowed without a *struct* hyperedge in between). If so, the E_H is simply disposed of, letting its children to be absorbed by its parent. By applying $E^{Doc}_{Doc\_Com}.Flatten()$ to `Design 3`, we obtain `Design 4`, where the `comments` are directly embedded inside each `product` without an enclosing `comments` document. However, the prefix `gotCom` followed by the counter still needs to be included in the JSON to distinguish different instances as shown in Fig. 8. Similarly, by applying $E^{Doc}_{List\_Com}.Flatten()$ on `Design 5`, we can flatten the list of `comments` into an embedded sequence with a counter.

The *group* transformation creates a E^{Doc}_{Set} around a child E^{Doc}_{Doc}(s) within another E^{Doc}_{Struct}. Both the child E^{Doc}_{Doc} and the defining path of E_Rs to it must already be inside the original E^{Doc}_{Struct}. By $E^{Doc}_{Top\_Prod}.group(E^{A\_ID,B\_ID}_{R}, E^{Doc}_{Struct\_gotCom})$ in `Design 3`, we obtain `Design 5` which embeds the `comments` inside each `product` document. Afterward, the child E^{Doc}_{Doc} and the E_Rs that are no longer used in any path to the remaining children, are removed from the original E^{Doc}_{Struct}. The transformation results in a new $List$ inside the JSON containing the data in the child E^{Doc}_{Doc} which is linked to the container by the path of E_Rs. To revert this transformation, we can call $flatten$ on the created E^{Doc}_{List} moving us back to `Design 3`. This transformation is illustrated as JSON in Fig. 9

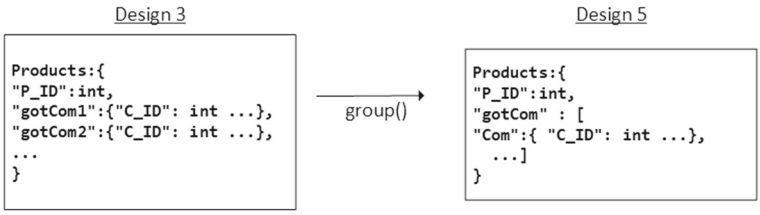


Fig. 9 Group transformation

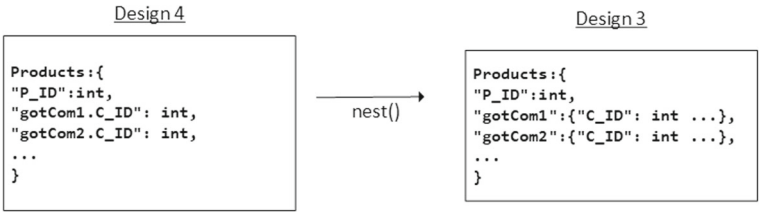


Fig. 10 Group transformation

The *nest* transformation creates a new embedded E^{Doc}_{Doc} inside another E^{Doc}_{Doc} (or E^{Doc}_{Top}). It is necessary to use the same parameters to create any document, but they must all be contained within the original E^{Doc}_{Doc}. After creating the embedded E^{Doc}_{Doc}, all its nodes are removed from the original E^{Doc}_{Doc}, except the E_Rs needed to keep it connected. Thus, *nest* does not allow keep redundant As or E_Hs in the original E^{Doc}_{Doc}; if redundancy is required, a *split* transformation needs to be done beforehand. By calling *nest* in `Design 4` to nest the `comment`, we can obtain back `Design 3` as shown in Fig. 10.

The *split* transformation allows creating a sibling independent E^{Doc}_{Doc} with all or part of the content of another one, inside its parent (this can result in E^{Doc}_{Top} instead of E^{Doc}_{Doc} if the parent is actually a collection). Some or all of the nodes in the new E^{Doc}_{Doc} can be removed from the original. Thus, the parameters of the transformation (which must all be inside the original E^{Doc}_{Doc}) are both the contents of the new E^{Doc}_{Doc} and which elements out of these are removed from the original. As a result, both E^{Doc}_{Struct}s either share the same $root$, or there will be a path between the $root$ of the original E^{Doc}_{Doc} and that of the new one. Notice that parameters must be so that both resulting E^{Doc}_{Doc} satisfy the invariants, but there is still freedom to determine whether this path is at the end contained in the original, new, or both E^{Doc}_{Struct}s. In our example, by splitting the *gotCom* from `product` in `Design 4`, we can obtain back `Design 2` (the dashed arrow depending on the path the parameters determine). An example of the split transformation is shown in Fig. 11. Finally, the *segregate* transformation divides an E^{Doc}_{List} (or E^{Doc}_{Col}) containing multiple E^{Doc}_{Doc}s or As into two. The only condition is that the segregated nodes must be already contained in the original E^{Doc}_{List}. After the transformation, the E_Rs that are no longer used by any of the children inside the original E^{Doc}_{List} are removed from it together with the segregated E^{Doc}_{List}. As a result, the corresponding JSON will contain two independent $list$s (or collections if we are talking about E^{Doc}_{Col}) with E^{Doc}_{Doc} or A, whose contents will depend on the path to E^{Doc}_{Doc} or A from the parent of the original E^{Doc}_{Set}. By calling $E^{Doc}_{Col\_ABs}.segregate(E^{Doc}_{Top\_Com})$ on `Design 2`, we can obtain `Design 1`. Figure 12 illustrates the segregation in JSON.

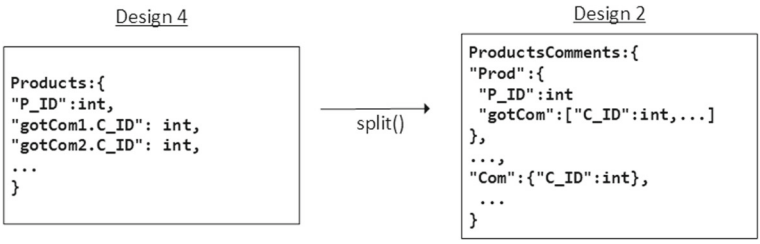


Fig. 11 Group transformation

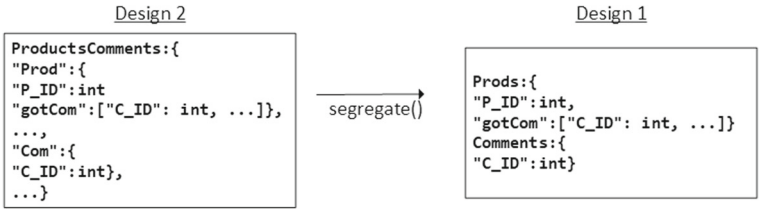


Fig. 12 Segregate transformation

Not having any normal forms or design algorithms to use as a baseline for comparison (like in RDBMS), we validate our document store design transformations against existing rule-based patterns. Luckily, MongoDB ones are publicly available.[5] Hence, we showcase our canonical hypergraph representation with MongoDB patterns and analyze how to implement them using a sequence of transformations in Appendix 1.

6 Experiments

We implemented our approach in a system called DocDesign 2.0 [10], using HypergraphDB[6] to store the canonical model, AIMA3e[7] for optimization using Java together with query cost estimator using Gekko[8] written in Python. In this section, we present its experimental evaluation, which is twofold. First, we analyze the quality of the designs generated (Sect. 6.1), and second, we evaluate the scalability of DocDesign 2.0 when the complexity of the entities and their relationships increase (Sect. 6.2).

6.1 Quality of the design

To evaluate the quality of the generated designs, we use DocDesign 2.0 on the running example of the RUBiS benchmark [11] (see Fig. 3). We prioritized query performance (0.7) followed by the storage space (0.2), depth of documents (0.05), and heterogeneity (0.05) together with the number of non-improving iterations (N in Algorithm,1) of 10. The generated design was then compared against the ones presented by the DBSR framework [5]. We used a higher weight for the query performance for our design to be comparable with DBSR while

[5] https://www.mongodb.com/blog/post/building-with-patterns-a-summary.

[6] http://www.hypergraphdb.org.

[7] https://github.com/aimacode/aima-java.

[8] https://gekko.readthedocs.io/en/latest.

Table 6 Workload used in the experiments

Query	Frequency
$\pi_{U\_ID}(\sigma_{B\_ID=b})$	4%
$\pi_{P\_ID}(\sigma_{B\_ID=b})$	2%
$\pi_{R\_ID}(\sigma_{U\_ID=u})$	4%
$\pi_{P\_ID}(\sigma_{U\_ID=u})$	6%
$\pi_{P\_ID}(\sigma_{U\_ID=u \wedge B\_ID=b})$	20%
$\pi_{U\_ID}(\sigma_{U\_ID=u \wedge B\_ID=b \wedge P\_ID=p})$	6%
$\pi_{C\_ID}(\sigma_{U\_ID=u})$	4%
$\pi_{C\_ID}(\sigma_{P\_ID=p})$	20%
$\pi_{U\_ID}(\sigma_{P\_ID=p \wedge C\_ID=c})$	8%
$\pi_{U\_ID}(\sigma_{P\_ID=p})$	6%
$\pi_{B\_ID}(\sigma_{P\_ID=p})$	20%

trying to improve storage size, and this configuration will be the typical ones that one would use where query performance is the most important aspect. Moreover, we have evaluated DocDesign 2.0 's capability to generate alternate designs depending on the weights provided by the user in our previous work [10].

We extended the DBSR evaluation benchmark to include the design suggested by DocDesign 2.0 . All the queries were executed using MongoDB Java driver 3.8.2. We used a single instance of MongoDB Community Edition version 4.2 running on Intel Xeon E5520, 24 GB of RAM with Debian 4.9 as the experimental setup. First, we generated data consisting of 1 million users, 10 million items, 5 million bids, 10 million comments, 3 million bids, and 4 regions. Then, the same data were stored in the alternative designs suggested by DBSR and DocDesign 2.0 . Next, 1 million random queries were executed, consisting of 11 different queries with their respective probabilities as shown in Table 6. The LHS of the arrow represents the selection, and RHS is the projection of the query. $P\_ID, C\_ID \rightarrow U\_ID$ means what is the `User` give a `Product` and a `Comment`. Finally, we measured the throughput of each of the alternate designs.

Table 7 shows the schemas of the designs generated by each of the systems. Designs generated by DBSR are based on joining the collections. Thus, the results can be controlled through the number of collections in the final design. In this scenario, we present the solutions of both 3 and 5 collections for comparison. It is clear that the designs generated by DBSR contain multiple redundancies, especially on the `product`. On the contrary, DocDesign 2.0 learns more toward having references and only embedding the `region` within the `user`. From an end-user perspective, the design from DocDesign 2.0 is much cleaner and has less maintenance compared to the ones of DBSR. Moreover, doing any updates will be pretty expensive in DBSR designs as it will involve updating multiple documents in different collections. The documents used in DBSR contain rather small documents and are unrealistic for a real-world scenario. Because of this, we conduct the same experiment with increased document sizes by converting the integer identifiers into MongoDB `UUID` fields (24 bytes instead of only 4) and increasing the description attribute size while maintaining the same workload.

Table 8 depicts the summary of the throughput values obtained for the 1 million random queries. The five-collection design of DBSR was also evaluated using the MongoDB aggregation framework. However, this has the worst performance out of all the designs. Since

Table 7 Final designs generated by DocDesign 2.0 and DBSR

DocDesign 2.0	DBSR (3 col)	DBSR (5 col)

"USER": {
"U_ID": int(4),
"U_F_NAME": varchar(20),
"REGION": {
"R_ID": int(4),
"R_NAME": varchar(10)
}
}
"PRODUCT": {
"P_ID": int(4),
"P_TITLE": varchar(10),
"BIDS": [{
"B_ID": int(4),
"B_PRICE": int,
"U_ID": int(4)
}],
"COMMENTS": [{
"C_ID": int(4),
"C_TITLE": varchar(20),
"U_ID": int(4)
}],
"U_ID": int(4)
}
	"BID-PRODUCT":{
	"B_ID": int(4),
	"B_PRICE": int,
	"U_ID": int(4),
	"PRODUCT": {
	"P_ID": int(4),
	"P_TITLE": varchar(10)
	}}
	"PRODUCT-SELLER-REGION": {
	"P_ID": int(4),
	"P_TITLE": varchar(10),
	"USER": {
	"U_ID": int(4),
	"U_F_NAME": varchar(20),
	"REGION": {
	"R_ID": int(4),
	"R_NAME": varchar(10)
	}}}
	"PRODUCT-COMMENTS": {
	"P_ID": int(4),
	"P_TITLE": varchar(10),
	"COMMENTS": [{
	"C_ID": int(4),
	"C_TITLE": varchar(20),
	"U_ID": int(4)
	}]}
		"PRODUCT-SELLER": {
		"P_ID": int(4),
		"P_TITLE": varchar(10),
		"USER": {
		"U_ID": int(4),
		"U_F_NAME": varchar(20)
		}
		}
		"PRODUCT-BIDS": {
		"P_ID": int(4),
		"P_TITLE": varchar(10),
		"BIDS": [{
		"B_ID": int(4),
		"B_PRICE": int,
		"U_ID": int(4)
		}]
		}
		+ DBSR (3 col)

Table 8 Performance comparison of the original dataset

	Runtime (ms)					
	Min	Q1	Mean	Media	Q3	Max
DocDesign 2.0	270	512	733	604	726	80,639
DBSR (3 col)	228	470	760	575	1074	80,063
DBSR (5 col)	262	523	787	585	2067	76,259
DBSR (5 col agg.)	253	542	1304	607	2471	75,583

Table 9 Performance comparison of the realistic dataset

	Runtime (ms)					
	Min	Q1	Mean	Media	Q3	Max
DocDesign 2.0	650	1121	1842	1268	2003	88,869
DBSR (3 col)	519	1292	2629	1782	4091	115,290
DBSR (5 col)	639	1558	2683	1866	7317	97,611
DBSR (5 col agg.)	619	1555	4147	1814	9587	108,083

DBSR can answer most of the queries with a single collection, the minimum runtime is lower as it is the time taken to retrieve the smallest cached document. The min runtime per query is highest on DocDesign 2.0 in both original and the large document experiment due to the smallest document being larger than the ones of DBSR (38 bytes *user* in DocDesign 2.0 vs 26-byte *bid-product* in DBSR).

However, DBSR loses the advantage when looking at the other statistics. In particular, the design with five collections falls quite behind; this could be mainly because the collections are competing on the available memory and a higher proportion of documents need to be fetched from the disk rather than the memory. In the original experiment, DocDesign 2.0 has a slight

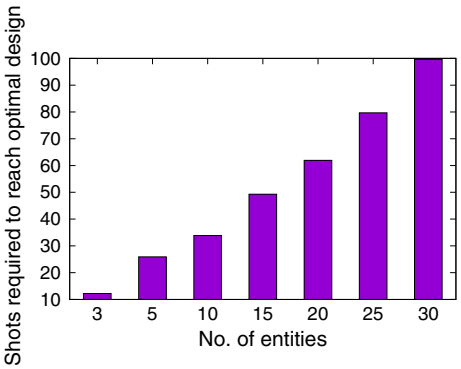


Fig. 13 Scalability of DocDesign 2.0 with number of entities

advantage in the mean and a higher one at Q3. However, once we increase the document size to be more realistic with the same workload (Table 9), DBSR always falls behind the performance of DocDesign 2.0 . Overall DocDesign 2.0 has better average performance and less skewed results by looking at the inter-quartile range, especially with larger document sizes. The maximum value of almost all the designs is quite similar because these queries involve fetching documents from the disk in the event of a cache miss. Both DBSR and DocDesign 2.0 did not generate heterogeneous designs as the optimal ones. In the case of DBSR, this is never considered, and in DocDesign 2.0 , since we are optimizing for query performance, the designs with heterogeneous collections become non-optimal. DocDesign 2.0 designs are less complex with a maximum depth of one level of nesting, while DBSR has two levels of nesting in the case of product-seller-region collection. When it comes to the total storage space, DBSR required 7GB in the three collections and 12GB in the case of five collections (due to the high redundancy in the generated designs of DBSR) as opposed to DocDesign 2.0 that required only 6.5 GB. Thus, it is clear that DocDesign 2.0 is capable of generating document store designs with better performance and superior space optimization.

6.2 Scalability of the approach

We tested the scalability of our approach in DocDesign 2.0 as it is an essential factor in larger use cases. In order to achieve this, we measured how many iterations it would take to get the near-optimal solution when the number of entities and relationships grows. For that purpose, we needed random ER diagrams with a varying number of entities as well as differentiate the topologies for each number of entities. This eliminates any opportunity of the topology affecting the final outcome of the experiment. We generated a synthetic ER diagram since it is impossible to find real-world ER diagrams that satisfy this requirement. Thus, we used gMark [26] (a graph instance and query workload generator) to create random ER topologies. We used a pre-defined number of entities and Gaussian distribution ($\mu = 0.31$ and $\sigma = 1$) of the relationships to generate the gMark graph and transformed it into our immutable graph.

Next, we generated as many random queries as the number of entities with equal probabilities. Finally, we used these values as input to DocDesign 2.0 to find the optimal design. We measured the number of iterations until there is no improvement for the next 100, assuming that this would give us the closest to the optimal solution. For each number of entities (experiment), we generated 10 random topologies, and for each topology ran DocDesign 2.0 10 times to obtain the average. As shown in Fig. 13, the number of shots required increases linearly as the number of entities grows. We appreciate that it requires around 100 (exactly

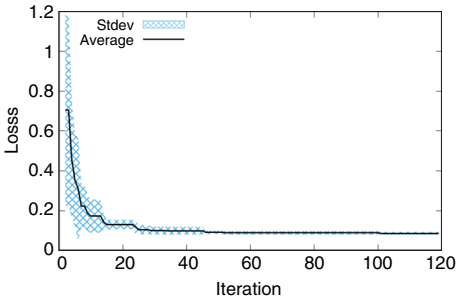


Fig. 14 Improvement over the number of shots

99.68) iterations to completely stabilize the design with 30 entities. However, in reality, one can obtain a near-optimal solution with much fewer iterations.

Figure 14 shows the evolution of the loss function that DocDesign 2.0 makes as the iterations (shots) progress in the experiment with 30 entities. The average and the standard deviation are of the 100 instances (10 topologies 10 runs) mentioned before. The initial shots make significant improvements fast, but as the shots progress, the improvement is minimal. Thus, we can safely assume that it is possible to obtain a near-optimal solution for this problem in around 15 iterations (i.e., N = 15 in Algorithm 1).

In summary:

- The design generated by using weights that represent typical requirement of optimizing queries with consideration on storage space outperforms the design generated by DBSR.
- DocDesign 2.0 's design was not only performant, but also requires less storage space.
- DocDesign 2.0 's multicriteria-based approach provides flexibility for the end users to optimize according to their requirements.
- The non-improving iterations (N) determine the optimality of the design. Through a synthetic workload with 30 entities with varying topology, we concluded that $N = 15$ would already generate a near-optimal solution.

6.3 Threats to validity

The first threat to validity is the possible bias on evaluating the quality of the design. The authors perceived having large collections with redundant data as a negative property of a design in comparison with DBSR. Nevertheless, we assume that this is the perspective of a traditional relational database designer in most of the cases as we experienced through the answers to the questionnaire we mentioned in the introduction. Only 9% of the responses chose redundant nested options as an optimal one.

Another threat to validity is that DocDesign 2.0 only provides a near-optimal solution and we do not know how far it really is from the best solution to a given problem. This can only be answered by testing all the possible implementations through an exhaustive search. This is a typical situation highly complex optimization problems. Thus, we used DBSR as a baseline to compare our solution instead.

An external threat to validity is the use of synthetic data to test the scalability of the approach instead of real-world data. We decided to use synthetic data because it allows to have full control over the scale factors of the input ER diagrams. Finding real-world data with such specific requirements is impossible.

Finally, the experimentation results might be biased in Table 9 where we increased the size of the dataset based on the fact that the design proposed by DocDesign 2.0 was not

outperforming on all the aspects (Q1 and Max) with the original dataset. However, in all the other aspects DocDesign 2.0 outperformed DBSR.

7 Conclusions

The popularity of document stores has prevailed among the plethora of NoSQL systems due to its flexible semi-structured data models. However, this flexibility poses a new challenge of determining the optimal data design for a particular use case. Currently, the data design is carried out in an trial-and-error or with simple ad-hoc rule-based approaches instead of a formal methodology such as normalization in RDBMS. The encouraged de-normalization through nesting and data redundancy increases the number of potential designs exponentially. Moreover, the database design of a document store affects not only its query performance but also other criteria such as storage space, data redundancy, and complexity of the stored documents. Thus, we approach the data design problem for document stores as a multicriteria optimization where we try to obtain a near-optimal design out of potential ones with a given use-case and user preference. We consider four cost functions in evaluating designs: query performance, storage size, heterogeneity of the collections, and depth of documents. The end-users can provide weights for these functions according to their preference. Then, once the end-user specifies the entity relationship diagram and the query workload, we use the shotgun hill-climbing approach to generate, evaluate, and present the near-optimal design solution for that particular use case.

To achieve this, first, we introduce and formalize a hypergraph-based canonical model to represent design alternatives. Then, we present an algorithm to generate random designs and an algebra of transformations to modify a database design to optimize it systematically. Finally, we propose a near-optimal design using a shotgun hill-climbing algorithm driven by the cost functions and the design operations. We evaluate our approach against DBSR, an automated database design solution purely driven by query cost. Our proposed design has a better performance than the competition and produces compact designs instead of the redundant ones given by DBSR. Moreover, we show that our approach scales up to provide the near-optimal design within 15 iterations of shotgun hill-climbing, even for complex use cases with the growth of entities and relationships.

This work is an initial step toward a novel approach for database designs for document stores that goes beyond simple query-based optimization. For future work, it would be of interest to include additional objective functions to be optimized such as number of nested collections, degree of redundancy, referencing rate, and width of the stored documents. However, it is necessary to identify if any correlations exist between them as this may push the final solution toward a specific area in the solution space, especially with positive correlations. Currently all the loss functions are calculated after each transformation. Therefore, it is possible to further optimize the execution of DocDesign 2.0 by incrementally updating possible loss functions together with the transformations. Moreover, it would also be of interest to see the possibility of pruning the search space during the exploration of parameters of transformations.

Acknowledgements This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence—Doctoral College" (IT4BI-DC). Sergi Nadal is partly supported by the Spanish Ministerio de Ciencia e Innovación, as well as the European Union—NextGenerationEU, under project FJC2020-045809-I / AEI/10.13039/501100011033.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate "Information Technologies for Business Intelligence—Doctoral College" (IT4BI-DC). Sergi Nadal is partly supported by the Spanish Ministerio de Ciencia e Innovación, as well as the European Union—NextGenerationEU, under project FJC2020-045809-I / AEI/10.13039/501100011033.

Data Availability Not applicable.

Code Availability https://github.com/modithah/CostCalculator.

Declarations

Conflict of interest The authors have no conflicts of interest to declare that are relevant to the content of this article.

Appendix A: Algorithm to build hyperedges from connected components

Algorithm 4 *BuildHyperedge* Algorithm

Input: Cnode *node*, HyperG *G*, List<A_C> *allAtoms*

```
list elements ← newList()
for all child ∈ node.children do
    elements.addAll(BuildHyperedge(child, G, allAtoms))
end for
if node.type = SKIP then
    elements.add(node.rel)
else if node.type = REF then
    elements.add(node.to)
    elements.add(node.rel)
else if node.type = NEST then
    hyp ← G.newHyperedge(DOCUMENT, node.to, {attributesrelationships(node.to), elements})
    elements.clear()
    elements.add(node.rel)
    elements.add(hyp)
    set ← G.newHyperedge(LIST, elements)
    elements.clear()
    elements.add(set)
    allAtoms.remove(node.to)
else if node.type = ROOT then
    hyp ← G.newHyperedge(TOPDOC, node.to, {attributesrelationships(node.to), elements})
    G.newHyperedge(COLLECTION, hyp)
    elements.clear()
    allAtoms.remove(node.to)
end if
return elements
```

Algorithm 4 generates corresponding E_H^{Doc}s recursively for a given connected component tree. It takes a $Cnode$, the hypergraph, and a list of unused A_Cs from Algorithm 2. We use a recursive call to build hyperedges to all the child $Cnode$s of a given $Cnode$ (lines 2–4) and keep them in a list of elements. Then, if the $Cnode$ type is *SKIP*, we only add the E_R of the $Cnode$ to the elements (lines 5–6). If it is *REF*, we add both E_R and its originating A_C (to) to the elements (lines 7–9). If it is *NEST*, we make a new E_{Doc}^{Doc} with the originating A_C (to) as the root, the A_As connected to the root and their corresponding E_Rs, and the built-up elements (line 11). Once we clear the elements in line 12, we build a E_{List}^{Doc} containing the newly created E_{Doc}^{Doc} and the E_R of the $Cnode$ (lines 11–15). Then, we reset the element list to contain only the new E_{List}^{Doc} and take out the originating A_C from the unused A_Cs in lines 16–18. If the $Cnode$ is $ROOT$, we build a E_{Top}^{Doc} with the element list and an E_{Col}^{Doc} containing the new E_{Top}^{Doc}, clear the element list, and remove the originating A_C from the unused A_Cs (lines 19–24). Finally, the collected element list is returned to be used by the calling function (line 25).

Appendix B: Formalized transformations

Formal definitions of the transformations discussed in Chapter 5.2 are described in Table 10. They allow to transform any valid document design and at the same time **guarantee the validity** of the resulting design. We use an auxiliary function $findRelPath(x : \mathbb{E}_H, y : \mathbb{E}_H)$ which will find the path of relations from x to $O(y)$.

Appendix C: Validation of operations against MongoDB design patterns

In the following, we go through each of the patterns, digest them, and use the same examples for illustration.[9] Nevertheless, our design transformations are defined at the logical level, so some physical patterns (e.g., involving indexing) cannot be fully represented.

Attribute pattern (Fig. 15) tries to identify a subset of fields that share some common characteristics that are frequently queried together, and add them into an array (e.g., release dates of the movies in different regions). The original document without the array has the field name suffixed by the region name, but since our immutable graph does not contain details about the instances, we simply use a counter instead as shown on `Design 4` in Fig. 5. Our set of transformation can implement this pattern by first using *nest* to create a flat embedded document and then using *group* to make a list out of the documents.

Bucket pattern (Fig. 16) groups specific data together (e.g., time series of sensor data). This pattern can be easily represented through our transformations by expliciting intervals a priori in the form of different $atom$s and $Relationship$s, since the predicates used for the bucket arrangement (e.g., start and end dates) cannot be generated as they are not part of the immutable graph. Moreover, the immutable graph must contain an additional A_C to identify each of the measurements. With that information, we can firstly use a sequence of *split*, *embed*, and *flatten* operations to change the root of the document (e.g., from measurement to sensor); then, similar to the attribute pattern, through *nest* and *group* we can create an embedded list with all measurements of the same sensor.

[9] See https://www.essi.upc.edu/~moditha/transformations.

Table 10 Document store-specific transformation methods

Method	⟨⟨preconditions⟩⟩	Activity

$E^{Doc}_{Struct}.nest($ $r : A_C,$ $Re : Set\ of\ E_R,$ $At : Set\ of\ A,$ $Do : Set\ of\ E^{Doc}_{Doc},$ $Li : Set\ of\ E^{Doc}_{List})$	• $r \in self$	$new\ Document(self.C, r, Re, At, Do, Li, self)$
		$keep \leftarrow \varnothing$
		for all $c \in (self.children \cap \mathbb{E}^{Doc}_{Doc}) - Do$ **do**
		$keep \leftarrow keep \cup FindRelPath(self, c)$
		end for
		for all $c \in (self.children \cap \mathbb{E}^{Doc}_{List}) - Li$ **do**
		for all $g \in (c.children \cap \mathbb{E}^{Doc}_{Doc})$ **do**
		$keep \leftarrow keep \cup FindRelPath(self, g)$
		end for
		end for
		for all $n \in ((Re \cup At \cup Do \cup Li) - keep)$ **do**
		$self.removeNode(n)$
		end for

$E^{Doc}_{Struct}.group($ $Re : Set\ of\ E_R,$ $s : E^{Doc}_{Doc} \mid A)$	• $Re \subset self$
	• $s \in self$
		if $s \in \mathbb{E}^{Doc}_{Struct}$ **then**
		$new\ List(self.C, Re, \{s\}, \varnothing, self)$
		else
		$new\ List(self.C, Re, \varnothing, \{s\}, self)$
		end if
		$self.removeNode(s)$
		$used \leftarrow \varnothing$
		for all $c \in self.children \cap (\mathbb{E}^{Doc}_{Doc} \cup \mathbb{A})$ **do**
		$used \leftarrow used \cup FindRelPath(self, c)$
		end for
		for all $r \in (Re - used)$ **do**
		$self.removeNode(r)$
		end for

$E^{Doc}_{Struct}.embed($ $: E^{Doc}_{Struct})$	• $self.parent = s.parent$
	• $O(self) \neq O(s) \Rightarrow$ $(\exists\{E^{x_1,x_2}_R, \ldots, E^{x_n,O(s)}_R\} \subseteq self \wedge x_1 \in self) \vee$ $(\exists\{E^{y_1,y_2}_R, \ldots, E^{y_n,O(self)}_R\} \subseteq s \wedge y_1 \in s)$
		$keep \leftarrow \varnothing$
		for all $c \in (self.parent.children \cap \mathbb{E}^{Doc}_{Struct}) - s$ **do**
		$keep \leftarrow keep \cup FindRelPath(self.parent, c)$
		end for
		for all $r \in FindRelPath(self.parent, s) - keep$ **do**
		$self.parent.removeNode(r)$
		end for
		$self.parent.removeNode(s)$
		$self.addNode(s)$

$E^{Doc}_{Struct}.split($ $r : A_C,$ $Re : Set\ of\ E_R,$ $At : Set\ of\ A,$ $Do :$ $Set\ of\ E^{Doc}_{Struct},$ $Li : Set\ of\ E^{Doc}_{Set},$ $Rm : Set\ of\ E_N)$	• $r \in (self \cap At)$
	• $Rm \subseteq (Re \cup At \cup Do \cup Li) - O(self) \subset self$
	• $O(self) \neq r \Longrightarrow$ $(\exists\{E^{x_1,x_2}_R, \ldots, E^{x_n,r}_R\} \subseteq (self - Rm)) \wedge x_1 \in (self - Rm)) \vee$ $(\exists\{E^{y_1,y_2}_R, \ldots, E^{y_n,O(self)}_R\} \subseteq Re \wedge y_1 \in At)$
		if $self \in \mathbb{E}^{Doc}_{Top}$ **then**
		$new\ TopDoc(self.C, r, Re, At, Do, Li, self.parent)$
		else
		$Rels \leftarrow FindRelPath(O(self), r)$
		for all $re \in Rels$ **do**
		$self.parent.addNode(re)$
		end for
		$new\ Document(self.C, r, Re, At, Do, Li, self.parent)$
		end if
		for all $n \in Rm$ **do**
		$self.removeNode(n)$
		end for

$E^{Doc}_{Set}.segregate($ $s : E^{Doc}_{Struct} \mid A)$	• $s \in self$	**if** $self \in \mathbb{E}^{Doc}_{Col}$ **then**
		$new\ Collection(self.C, s)$
		else
		$keep \leftarrow \varnothing$
		$Re \leftarrow FindRelPath(self.parent, s)$
		for all $c \in (self.children - s)$ **do**
		$keep \leftarrow keep \cup FindRelPath(self.parent, c)$
		end for
		if $s \in \mathbb{E}^{Doc}_{Doc}$ **then**
		$new\ List(self.C, Re, \{s\}, \varnothing, self.parent)$
		else
		$new\ List(self.C, Re, \varnothing, \{s\}, self.parent)$
		end if
		for all each $r \in (Re - keep)$ **do**
		$self.removeNode(r)$
		end for
		end if
		$self.removeNode(s)$

$E^{Doc}_{Set}.union(s$ $: E^{Doc}_{Set})$	• $self.parent = s.parent$	$self.addNode(s)$ $self.parent.removeNode(s)$ $s.dispose()$
$E^{Doc}_{H}.flatten()$	• $self.parent \in \mathbb{E}^{Doc}_{Struct}$	$self.dispose()$

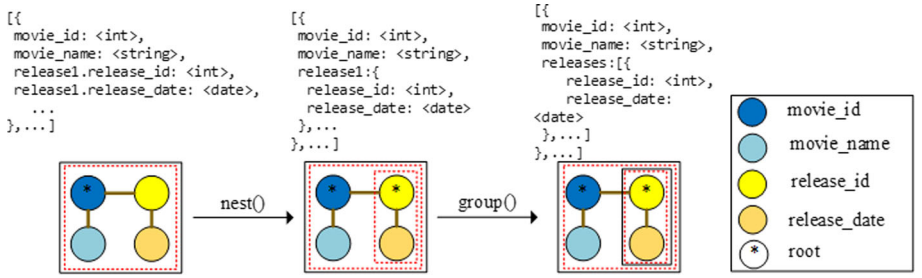


Fig. 15 Transformations of the **Attribute** pattern

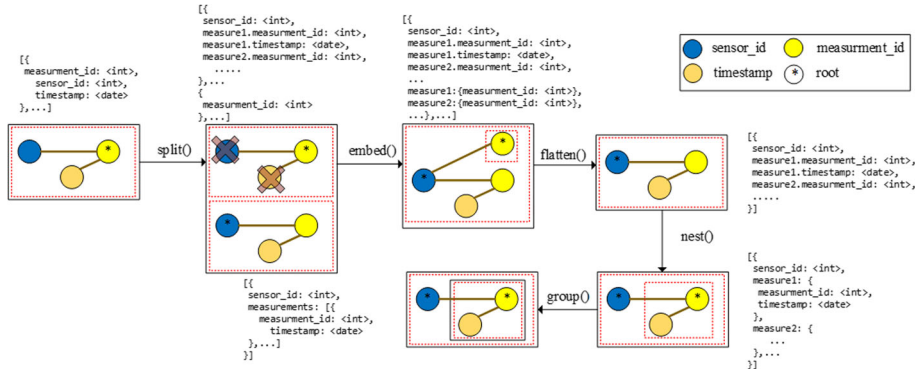


Fig. 16 Transformations of the **Bucket** pattern

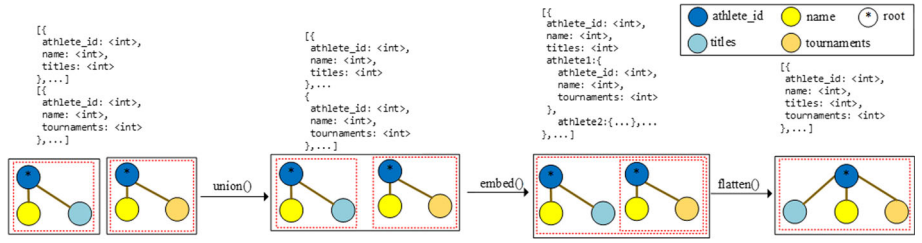


Fig. 17 Transformations of the **Polymorphic** pattern

Polymorphic pattern (Fig. 17) is used to merge collections with documents that share multiple attributes (e.g., Bowling and Tennis Athletes). We can easily deal with such transformation by *union*, *embed*, and *flatten* operations. However, currently, our hypergraph does not support specialization due to the complexity of guaranteeing the validity of the design (i.e., ensuring none of the subclasses/partitions are missed as a result of the transformations). Thus, this pattern can only be partially represented. However, it is possible to fully represent if the immutable graph contained the subclass information.

Extended reference pattern (Fig. 18) is a mean of avoiding joins by embedding frequently accessed data of two entities (e.g., customer address in an order). First, we use *split* to extract from one document the information that needs to be embedded in the other (e.g., the address

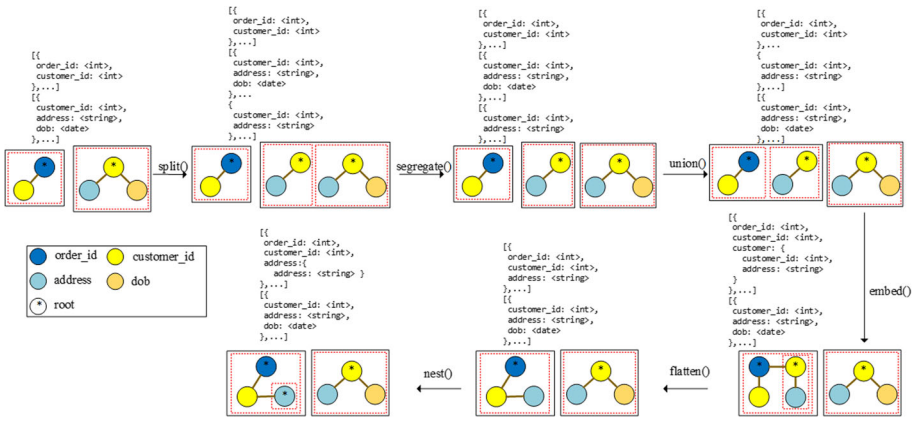


Fig. 18 Transformations of the **Extended reference** pattern

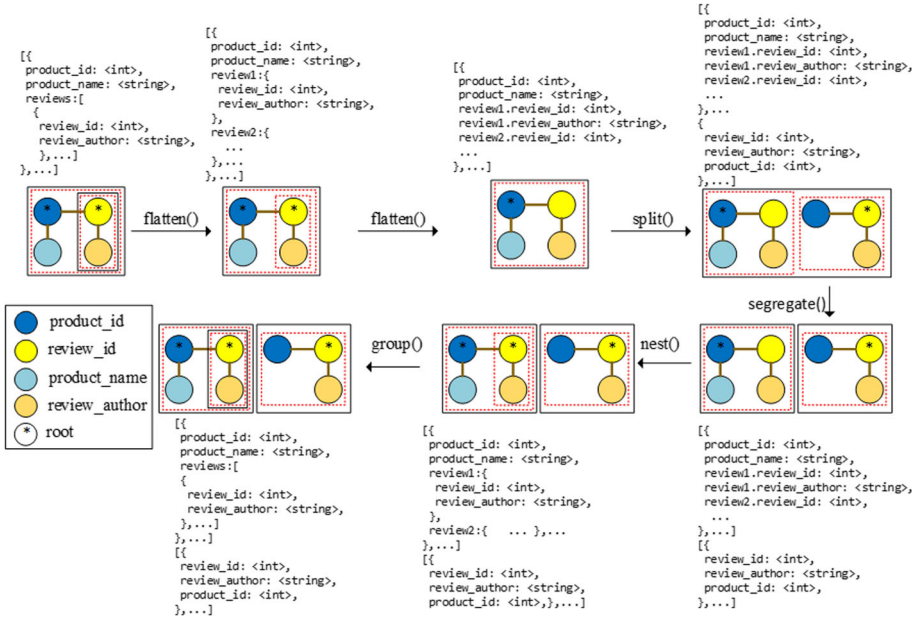


Fig. 19 Transformations of the **Subset** pattern

from the customer) and *segregate* this into a new collection. Then, we *union* the new collection and the one that requires embedding, to finally, join the information by *embed*, *flatten*, and *nest*.

Subset pattern (Fig. 19) is used to prevent unnecessary growth of documents. A typical example would be to only store the first n documents in a list, keeping the rest of the list in a separate collection. As in the bucket pattern, n should be encoded somehow in the immutable graph (e.g., representing the two sets of documents in different classes). First, we remove the list with two consecutive *flatten* operations. Next, we *split* the document and *segregate* the

part that needs to be moved to a new collection. Finally, we *nest* and *group* to recreate the original list.

The remaining seven MongoDB design patterns (namely **Outlier**, **Approximation**, **Computed**, **Document versioning**, **Preallocated**, **Schema versioning**, and **Tree and graph**) can be represented in our canonical model, provided the immutable graph contains the required information (e.g., schema/document version, the average of an attribute), but besides that, they require changes in the client application logic or the engine configuration rather than in the document design. Thus, they are out of the scope of this work.

References

1. Cattell R (2010) Scalable SQL and NoSQL data stores. SIGMOD Rec 39(4):12–27
2. D'mello BJ, Satheesh M, Krol J (2007) Web development with MongoDB and node, 3rd edn. Packt Publishing, Birmingham
3. Atzeni P, Bugiotti F, Cabibbo L, Torlone R (2020) Data modeling in the NoSQL world. Comput Stand Interfaces 67:103149. https://www.sciencedirect.com/science/article/abs/pii/S0920548916301180
4. Graham RL, Knuth DE, Patashnik O (1994) Concrete mathematics: a foundation for computer science, 2nd edn. Addison-Wesley, Boston
5. Reniers V, Van Landuyt D, Rafique A, Joosen W (2020) A workload-driven document database schema recommender (DBSR). In: International conference on conceptual modeling. ER, pp 471–484
6. Mior MJ, Salem K, Aboulnaga A, Liu R (2017) NoSE: schema design for NoSQL applications. IEEE Trans Knowl Data Eng 29(10):2275–2289
7. de la Vega A, García-Saiz D, Blanco C, Zorrilla ME, Sánchez P (2020) Mortadelo: automatic generation of NoSQL stores from platform-independent data models. Future Gen Comput Syst 105:455–474
8. Hewasinghage M, Abelló A, Varga J, Zimányi E (2020) DocDesign: cost-based database design for document stores. In: International conference on scientific and statistical database management. SSDBM, pp 27–1274
9. Cho J, Wang Y, Chen I, Chan KS, Swami A (2017) A survey on modeling and optimizing multi-objective systems. IEEE Commun Surv Tutor 19(3):1867–1901
10. Hewasinghage M, Nadal S, Abelló A (2021) Docdesign 2.0: automated database design for document stores with multi-criteria optimization. In: International conference on extending database technology, EDBT, pp 674–677
11. Cecchet E, Marguerite J, Zwaenepoel W (2002) Performance and scalability of EJB applications. In: ACM SIGPLAN conference on object-oriented programming systems, languages and applications, OOPSLA, pp 246–261
12. Pezoa F, Reutter JL, Suárez F, Ugarte M, Vrgoc D (2016) Foundations of JSON schema. In: International conference on the World-Wide Web, WWW, pp 263–273. https://doi.org/10.1145/2872427.2883029
13. Klettke M, Störl U, Scherzinger S (2015) Schema extraction and structural outlier detection for JSON-based NoSQL data stores. In: Datenbanksysteme Für Business, Technologie und Web. BTW, pp 425–444
14. Kanade A, Gopal A, Kanade S (2014) A study of normalization and embedding in MongoDB. In: IEEE international advanced computing conference. IACC, pp 416–421
15. Hewasinghage M, Nadal S, Abelló A (2020) On the performance impact of using JSON, beyond impedance mismatch. In: New types of applications and information systems, pp 73–83
16. Gómez P, Roncancio C, Casallas R (2018) Towards quality analysis for document oriented bases. In: International conference on conceptual modeling, ER, pp 200–216
17. Herrero V, Abelló A, Romero O (2016) NoSQL design for analytical workloads: variability matters. In: International conference on conceptual modeling, ER, pp 50–64
18. Soransso RASN, Cavalcanti MC (2018) Data modeling for analytical queries on document-oriented DBMS. In: ACM symposium on applied computing. SAC, pp 541–548
19. Hewasinghage M, Abelló A, Varga J, Zimányi E (2021) Managing polyglot systems metadata with hypergraphs. Data Knowl Eng 134:101896
20. Atzeni P, Bugiotti F, Rossi L (2012) Uniform access to non-relational database systems: The SOS platform. In: International conference on advanced information systems engineering. CAiSE, pp 160–174
21. Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering

22. Hewasinghage M, Abelló A, Varga J, Zimányi E (2021) A cost model for random access queries in document stores. VLDB J 30(4):559–578
23. de Lima C, dos Santos Mello R (2015) A workload-driven logical design approach for NoSQL document databases. In: International conference on information integration and web-based applications & services. iiWAS, pp 73–17310
24. Chebotko A, Kashlev A, Lu S (2015) A big data modeling methodology for apache Cassandra. In: IEEE international congress on big data, pp 238–245
25. Marler RT, Arora JS (2004) Survey of multi-objective optimization methods for engineering. Struct Multidiscip Optim 26(6):369–395
26. Bagan G, Bonifati A, Ciucanu R, Fletcher GHL, Lemay A, Advokaat N (2017) gmark: schema-driven generation of graphs and queries. IEEE Trans Knowl Data Eng 29(4):856–869

Moditha Hewasinghage is a data engineer and researcher focusing on Big Data and Data Science areas. He holds a joint Ph.D. degree from Universitat Politècnica de Catalunya (UPC) and Université Libre de Bruxelles (ULB) through the IT4BI-DC Erasmus Mundus Joint Doctorate. His research interests include data modeling, query optimization, cost models, and document stores.

Sergi Nadal completed the Ph.D. in Computer Science in 2019 from Universitat Politècnica de Catalunya (UPC) and Université Libre de Bruxelles (ULB). He is a Juan de la Cierva Formación postdoctoral fellow and teaching assistant in the Database Technologies and Information Management (DTIM) group in UPC. His research interests lie on system aspects of data and information management.

Alberto Abelló is an Associate professor and completed Ph.D. in Informatics, UPC. He is a Local coordinator of the Erasmus Mundus Ph.D. program IT4BI-DC and MSCA-ITN-EJD DEDS. He is a active researcher with more than 100 peer-reviewed publications and H-factor of 31 according to Google Scholar. His interests include Data Warehousing and OLAP, Ontologies, NOSQL systems, and BigData management. He has served as Program Chair of DOLAP and MEDI, being member also of the PC of other database conferences like DaWaK, CIKM, VLDB, etc.

Esteban Zimányi is a professor and a director of the Department of Computer and Decision Engineering (CoDE) at Université libre de Bruxelles. He co-authored and co-edited 23 books and numerous peer-reviewed publications. He was Editor-in-Chief of the Journal on Data Semantics published by Springer from 2012 to 2020. He has been coordinator of the IT4BI, IT4BI-DC, and BDMA, and DEDS European Master and Doctorate programs. His current research interests include data warehouses, spatiotemporal databases, geographic information systems, and semantic web.