CONTENTS

DocDesign 2.0: Automated Database Design for Document Stores with Multi-criteria Optimization

Moditha Hewasinghage, Sergi Nadal, Alberto Abelló
Universitat Politècnica de Catalunya
Barcelona, Spain
moditha|snadal|aabello@essi.upc.edu

ABSTRACT

We present DocDesign 2.0, a novel system that supports database design for document stores. DocDesign 2.0 automatically generates a document store design driven by a query workload and a set of optimization objectives. In the presence of a massive search space, DocDesign 2.0 adopts multi-objective optimization techniques that, with high probability, guarantee to yield the optimal design based on the preferences (i.e., weights) provided by the end-user. In this paper, we demonstrate how DocDesign 2.0 improves the productivity on the task of designing a document store, as well as how the quality of the results is improved with respect to those obtained by manually generating the design.

1 INTRODUCTION

The plethora of current NoSQL systems introduces alternative data storage methods to the traditional relational database management systems (RDBMSs) [2]. Among these, document stores have gained popularity due to the semi-structured data storage model. In contrast to the RDBMS normalization, document stores favor embedding, trying to keep the data related to a single instance together instead of spreading it across different tables. This increases the complexity of database design for document stores as opposed to RDBMS, where reaching 3NF or BCNF guarantees an optimal database design in the majority of the use-cases. Database design for document stores is, in general, given low precedence, and mostly carried out in a rule-based ad-hoc manner. For instance, MongoDB, the leading document store, provides a set of design patterns[1] that provide certain guidelines on how to structure documents. However, it has been shown that the choice of design has a major impact on performance, specially in the NOSQL realm [1]. Thus, it is advantageous to have a better design by exploiting any prior knowledge on the requirements rather than a purely random one.

Let us take an example of implementing an online auction system based on the RUBiS benchmark [3] in a document store. Fig. 1 shows the 5 entities and 6 relationships composing the
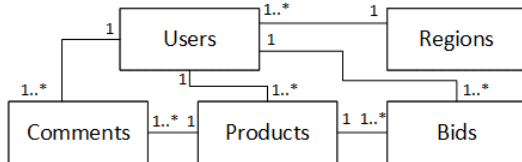

Figure 1: ER diagram of RUBiS Benchmark

---

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

RUBiS framework. We can have a normalized solution, similar to that in a RDBMS, an embedded single-document solution, or the solution suggested by a purely workload-based schema recommender, such as DBSR [11], which denormalizes certain entities. To show the complexity of finding the optimal database design in a document store, let us define a running example use case consisting of two single entities from RUBiS, namely Product and Comments, and an equiprobable hypothetical workload defined as follows:

- Given a Comment ID, find its text.
- Given a Product ID, find its name.
- Given a Comment ID, find the Product name.
- Given a Product ID, find all of its Comments.

In this scenario, we have two entities and one relationship. If we assume that all attributes for an entity are kept together within a document, we are left with the decision on where the relationship must be stored in the final design. Thus, database designs can be enumerated based on the alternatives to store the relationship, which depend on three independent choices: direction, representing, and structuring as shown in Fig. 2, together with two examples. **Direction** determines which entity keeps the information about the relationship. It can be one of the two entities, or both. **Representation** affects how this relationship is stored either by keeping a reference or embedding the object. Finally, **Structuring** determines how we structure the relationship, either as a nested list or flattened. For example, if keep the references to the comments in the product, they can
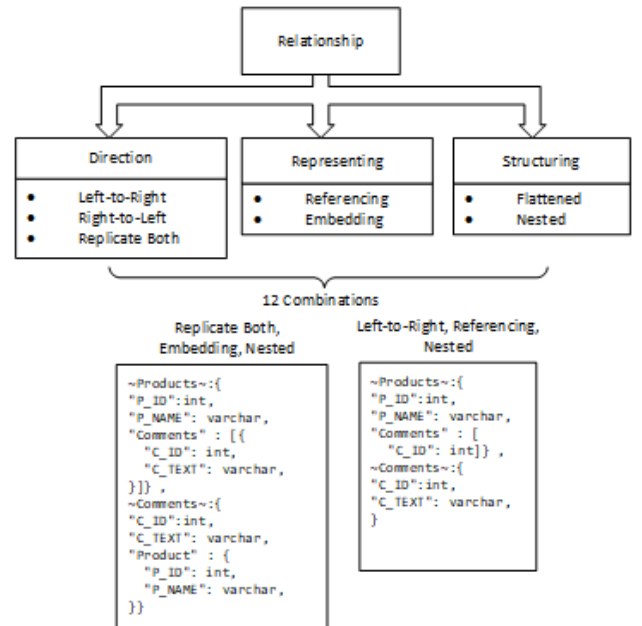

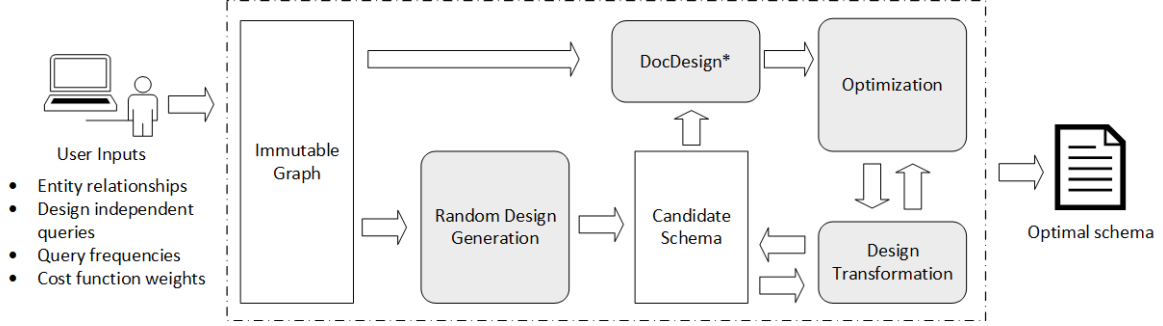**Figure 2: Relationship design choices, and two examples**

Figure 3: Overview of the DocDesign 2.0 Architecture

be stored as a list of references (*comment:...*) or in a flattened manner (*comment_1:.., comment_2:..*). Hence, we end up with 12 possible designs for our running example. Each of these could potentially be the optimal solution for an end-user depending on their preferences. For example, the design where products and comments nest their counterparts redundantly (i.e., both directions are stored by embedding the objects) will benefit query performance, as all queries can be answered with a single random access. However, this is at the expense of storage space due to redundancy. What if we only have a single reference on the product for its comments? Does the reduction of storage space justify the impact on performance? This trade-off between alternatives makes the process of finding the optimal design a complex one.

The number of relationships of the use-case (r) determines the number of candidate designs, which is exponential (12^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, user and region 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, used to calculate the number of ways to partition n distinct elements into k non-empty subsets [5]. Overall, such exponential growth makes impossible to enumerate and evaluate all candidate designs. Hence, existing solutions, such as DBSR [11], NoSE [10] and Mortadelo [4], mainly rely on the query workload to propose a database design.

Contributions. Considering the above observations it is clear that the problem of storage design for document stores has a large search space. Moreover, each candidate solution potentially performs differently among the considered cost functions. It is, hence, obvious that exhaustively exploring the search space is prohibitively expensive. To overcome this issues, in this paper, we present DocDesign 2.0, a novel solution that addresses the complex problem of database design for document stores. DocDesign 2.0's contributions involve *automatically generating potential designs*, as well as *evaluating the performance of a design on four objectives: storage size, query performance, degree of heterogeneity, and average depth of documents*. Finally, DocDesign 2.0 *presents the end-user with the near-optimal database design specific to his/her preference of the objective* for a given use-case and query workload. Precisely, in this paper, we consider read-only query workloads. DocDesign 2.0 embeds and extends our former solution DocDesign [7], which aids on evaluating database

designs based on storage size and query performance, requiring however to provide a concrete schema as input. Contrarily, DocDesign 2.0 automatically generates such designs yielding, with a high probability, the near-optimal one with respect to a set of objectives.

Outline. In the rest of the paper we introduce DocDesign 2.0's demonstrable features to resolve the motivational example and other database design for document stores scenarios. We first provide an overview of DocDesign 2.0and its core features. Lastly, we outline our on-site demonstration, involving the motivational scenario as well as other more complex real-world use cases.

2 DOCDESIGN 2.0 IN A NUTSHELL

DocDesign 2.0 adopts multi-objective optimization techniques, which have shown to be effective on obtaining near-optimal solutions out of a large search space in the presence of contradicting objectives [9]. In these scenarios, one can only aim to obtain a Pareto solution (a solution that, in the presence of multiple objectives, cannot improve one objective without worsening another).

Search algorithm. Local search algorithms consist of the systematic modification of a given state, by means of 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 DocDesign 2.0 can tackle, we decided to choose *hill-climbing*, a non-parametrized search algorithm which can be seen as a local search, always following the path that yields higher utility 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.

An overview of DocDesign 2.0 is shown in Fig. 3 and we present the modules and components of DocDesign 2.0 in the following subsections.

2.1 User Inputs

There are three inputs the end-user must provide, namely the equivalent to an Entity-Relationship diagram of the domain, query workload, and the weights of the cost functions.

Entity-Relationship. Refers to the use case-specific entities, their attributes, and the relationships between them. To accurately measure the different cost functions, DocDesign 2.0 requires the number of instances of each entity, the size of its
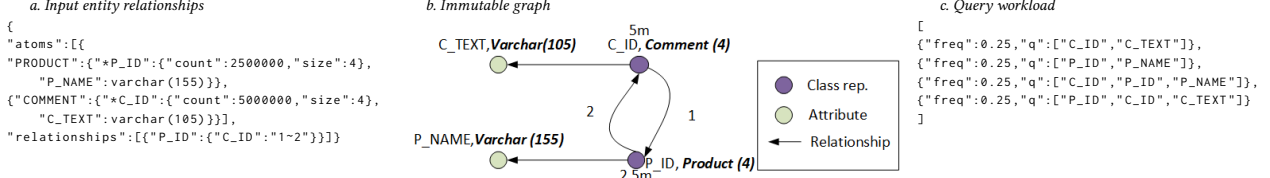
a. Input entity relationships
```
{
"atoms":[{
"PRODUCT":{"*P_ID":{"count":2500000,"size":4},
    "P_NAME":varchar(155)}},
{"COMMENT":{"*C_ID":{"count":5000000,"size":4},
    "C_TEXT":varchar(105)}}],
"relationships":[{"P_ID":{"C_ID":"1~2"}}]]
```

b. Immutable graph

c. Query workload
```
[
{"freq":0.25,"q":["C_ID","C_TEXT"]},
{"freq":0.25,"q":["P_ID","P_NAME"]},
{"freq":0.25,"q":["C_ID","P_ID","P_NAME"]},
{"freq":0.25,"q":["P_ID","C_ID","C_TEXT"]}
]
```

Figure 4: Input entity relationships, its internal graph representation, and query workload

attributes, and the relationship multiplicities (Fig. 4.a). This information is considered immutable, and the database design is carried out on top of it. We use a hypergraph-based canonical model internally to represent them [8]. (shown in Fig. 4.b). Furthermore, the entities are atomic, meaning that attributes related to an entity cannot be split.

Query workload. Consists of a set of queries together with their frequencies to be executed in the use case. These queries are independent of the database design and are represented as subsets of the immutable information (Fig. 4.c).

Cost function weights. Allows the end-user to include his/her preference in the database design. Currently, DocDesign 2.0 supports tuning four cost functions corresponding to the objectives: query cost, storage size, degree of heterogeneity within collections and sets, and average depth of the documents. The end-user can decide how important each of these costs are and resolve trade-offs between them. For instance, forcing higher importance to query cost and lowering the one of storage size would lead to a schema with higher redundancy and better performance.

2.2 Design Operations

Information about entities, their attributes, and the relationships are considered immutable, and the database design is built from it. Indeed, with regard to our running example, the final design must have information on all the warehouses, districts, and the relationships between them. A hypergraph-based representation enables DocDesign 2.0 to guarantee this property (we refer the reader to [7, 8] for further details). We introduce two methods to fit the shotgun hill climbing approach: generation of a random design, and evolution of a design using valid transformations.

Random design generation. The random schema generator relies on identifying subsets of entities and relationships that will be made into a collection (referred to as connected components) and the structure of the documents inside the collection in a document store database design. Based on the 12 possible designs that a relationship can be stored, we make the following decisions randomly in the schema generation process.

- **Root of the connected component** is chosen at random from the available entities. This choice determines the root document of the document store collection that this component represents. In our running example, this is either picking the warehouse or the district as the root of the collection. Let us assume we picked the warehouse in this case.
- **Choosing the next path to explore** expands the connected component and determines it's structure. Potentially multiple relationships connect an entity to others in a connected component. Thus, for a given entity of a connected component, a random subset of these relationships is picked to further expand, determining the depth and the related documents of the final design. This, together with the root of the document determines the choice of the direction in Fig. 2 except for replicating

both. In the running example, we choose the relationship to the district from the warehouse (already inside the component).
- **Embedding or Reference** determines possible ways to represent the relationship between two entities of a component. If embedding is chosen, the entire document is embedded in the parent and referencing only keeps the reference of the related document on the parent. In the running example, if the embedding option is chosen, the final collection will be warehouses with embedded districts. We also make the decision of replicating both based on a given probability.

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 with regard to the options in Fig. 2. 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.

Design transformations. Even though it is possible to generate most of the potential designs through the random generator, it is very unlikely to reach an optimal state randomly. Moreover, we omitted the heterogeneous collections/lists and flattened ones in the random process. Thus, we introduce seven design transformation operations and use five of them to generate the neighbors of a particular design. These transformations are inspired by the rule-based design patterns proposed by MongoDB.We have validated them by recreating the MongoDB design patterns as sequences of transformations[2].

- **Union** - merges two collections/lists at the same level and creates a heterogeneous one.
- **Segregate** - separates a homogeneous collection/list out of a heterogeneous one.
- **Embed** - embeds a related document inside another.
- **Flatten** - flattens an embedded document or a list inside it's parent.
- **Group** - creates an embedded list of related documents inside another (opposite of flattening a list).

We also identify two other operations, namely, **Nest** and **Split**. Nest operation creates a nested document inside another and is unnecessary as we already cover it through the random generation. Split is similar to vertical partitioning a document. However, adhering to the atomic entity rule, we decided not to include this operation as it would also expand the search space uncontrollably.

2.3 Optimization

Candidate designs obtained through random generation or transformation need to be evaluated in order to assess their optimality.

---

[2]More details at https://www.essi.upc.edu/~moditha/transformations
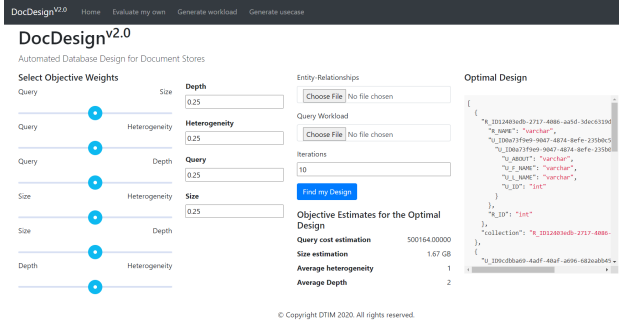
3

Figure 5: DocDesign 2.0 user interface

Cost functions. We introduce four cost functions to be measured and optimized in DocDesign 2.0: query cost, storage cost, degree of heterogeneity, and average depth of the documents. These are defined as follows:

- **Query cost** (CF_Q), is the sum of the relative query performance values calculated from the schema using a cost model for document stores [6].
- **Storage size** (CF_S), is the total storage size required by the collections and indexes, calculated using the canonical model.
- **Degree of heterogeneity** (CF_H), is the number of different types of documents in a collection/list. We use the average over all the collections and lists of the schema. Each heterogeneity is given a weight depending on which level the list/collection lies in the document. The higher the level, the higher the assigned weight, penalizing heterogeneities at higher levels of the document structure.
- **Depth of the documents** (CF_D), is the average depth of the documents of the design.

Utility function. Guiding the local search algorithm requires the definition of a utility function taking into account the end-user's preferences. Here, this is a function to be minimized. Hence, the end-user can assign weights to each of the cost functions according to their importance in the use-case. Then, for a given design C, we define the utility as the normalized weighted sum of each cost function $u(C) = \sum_{i=1}^{n} w_i \frac{CF_i(C) - CF_i^o}{CF_i^{max} - CF_i^o}$. The expression considers the weight w of each cost function, which is used on the transformed utility function for C. This is a normalized value that considers the *utopia* (i.e., the expected minimal) and the maximal design costs, yielding values between zero and one.

3 DEMONSTRATION OVERVIEW

DocDesign 2.0 has a web interface as shown in Fig. 5. In the on-site demonstration, we will showcase DocDesign 2.0 using the RUBiS usecase as a real-world example The manual database design process is expensive as RuBiS contains five entities and six relationships, leading to a large solution space. Moreover, we use the 11 queries with their access frequencies as the workload. First, for the ease of explanation, we will use the paper's running example (i.e., Products and Comments) and the four queries to showcase the ease of using DocDesign 2.0, initially with equal weights and then higher weight to query cost. In the first scenario with equal weights, the optimal schema is products having references to their comments. When optimizing only for the query performance, DocDesign 2.0 suggests redundantly

nesting comments inside the product and product inside the comment. This approach reduces the actual runtime almost by half at the expense of double the storage space. This establishes the functionality and the efficiency of DocDesign 2.0.

Then, we will import the full RUBiS E/R to DocDesign 2.0 together with the queries and showing the ability of DocDesign 2.0 to solve more complex use-cases. The results presented by DocDesign 2.0 have a higher throughput once implemented compared to the best solution suggested by DBSR [11]. Moreover, the suggestion by DocDesign 2.0 has far less redundancy compared to the ones by DBSR. The participants are also allowed to interact with the DocDesign 2.0 demonstration with the ability to choose between different queries and objective function weights as well as generate their own. The resulting updates made to the design can be discussed by means of changes introduced (e.g.: giving more importance to query cost will result data redundancy). We also present the actual runtimes (calculated by a benchmarking suite) and storage sizes for the usecases and the designs that we demonstrate. This allows the users to validate the effectiveness of the solutions generated by DocDesign 2.0.

Since the JSON input format is specific to DocDesign 2.0, we also include a functionality to create them through an intuitive UI. Moreover, the users can suggest their own design to compare against the one suggested in terms of the four objective functions. The designs suggested by DocDesign 2.0 rely on pre-defined queries. If the queries are unknown the end users have to rely on the other three cost functions to obtain a "good enough" design. Through this hands-on experience, we are able to show the ability of DocDesign 2.0 to address the complex problem of document store database design improving the quality and productivity as opposed to a manual design process.[3]

REFERENCES

[1] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. 2016. Data modeling in the NoSQL world. *Computer Standards & Interfaces* (2016).
[2] Rick Cattell. 2010. Scalable SQL and NoSQL data stores. *SIGMOD Record* 39, 4 (2010).
[3] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. 2002. Performance and scalability of EJB applications. In *SIGPLAN*. ACM, 246–261.
[4] A. de la Vega, D. García-Saiz, C. Blanco, M. E. Zorrilla, and P. Sánchez. 2020. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models. *Future Gen. Comp. Sys.* 105 (2020).
[5] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley.
[6] Moditha Hewasinghage, Alberto Abelló, Jovan Varga, and Esteban Zimányi. [n.d.]. A Cost Model for Random Access Queries in Document Stores (Under review).
[7] Moditha Hewasinghage, Alberto Abelló, Jovan Varga, and Esteban Zimányi. 2020. DocDesign: Cost-Based Database Design for Document Stores. In *Int. Conf. Scientific and Statistical Database Management*. SSDBM.
[8] Moditha Hewasinghage, Jovan Varga, Alberto Abelló, and Esteban Zimányi. 2018. Managing Polyglot Systems Metadata with Hypergraphs. In *Int. Conf. on Conceptual Modeling*. ER.
[9] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* 26, 6 (2004), 369–395.
[10] Michael Joseph Mior, Kenneth Salem, Ashraf Aboulnaga, and Rui Liu. 2017. NoSE: Schema design for NoSQL applications. *IEEE Trans. Knowl. Data Eng.* 29, 10 (2017).
[11] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. 2020. A Workload-Driven Document Database Schema Recommender (DBSR). In *Int. Conf. on Conceptual Modeling*. ER.

---

[3]Demo video available at https://vimeo.com/505248323