JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 479-496 (2022)
DOI: 10.6688/JISE.202203_38(2).0012

Aggregates Selection in Replicated Document-Oriented Databases

KHALED JOUINI
MARS Research Lab LR17ES05
ISITCom, University of Sousse
H. Sousse, 4011 Tunisia
E-mail: khaled.jouini@isitc.u-sousse.tn

Document-stores leverage the flexibility of structured documents to pack closely related data within a single autonomous *aggregate* (*i.e.* document). Selecting an appropriate set of aggregates for a document database is a non-trivial task since: (i) there are no clear-cut transformation rules from a conceptual design to a document design; (ii) a large space of design options must often be considered; and (iii) most importantly, it is difficult, if not impossible, to find out a single set of aggregates suitable for all data access patterns.

In a previous work, we proposed *distorted replicas*: a replication scheme that leverages ubiquitous replication in document-stores and restructures replicated data in different ways to better cope with the heterogeneity of data access patterns. In this paper, we tackle the problem of aggregates selection and replication in an integrated manner. In particular, given a database with a replication factor of C and a workload W, we propose novel cost-driven techniques allowing to: (i) determine the most interesting aggregates; and (ii) pack the most interesting aggregates into C disjoint and complete subsets in such a way that the execution time of W is minimized. Experimental results obtained over two real-world workloads showed that distorted replicas allow to run queries up to tens of times faster than state-of-the-art approaches.

Keywords: logical & physical design, aggregate data model, replication, 0-1 MKP, document-stores

1. INTRODUCTION

NoSQL systems rise has been driven by the desire to store data on large clusters of commodity servers and to provide horizontal scalability, high availability, and high throughput for write/read operations [1]. Document-stores leverage the flexibility of structured documents (*e.g.* JSON) to pack closely related data in a single autonomous document or *aggregate*, rather than having them scattered across several tables as in the relational model. By doing so, document-stores manipulate related data in a single database operation and avoid cross-nodes writes and joins, which are prohibitive in highly distributed environments [2]. Fig. 1 depicts a slightly modified JSON-formatted document from the archives of the DBLP bibliography [3] and illustrates the key differences between the aggregate data model and the relational data model.

Received March 16, 2020; revised August 26, 2020; accepted October 5, 2020.
Communicated by Reynold C.K. Cheng.

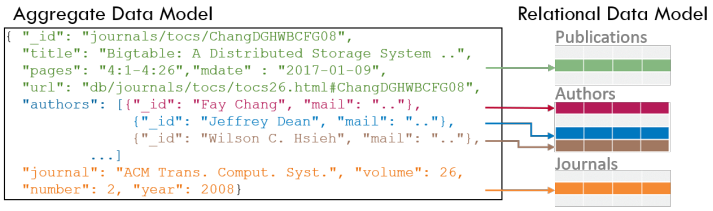

Fig. 1. Aggregate data model vs. Relational data model.

A key challenge in document-stores is how to model documents to meet the needs of applications in terms of performance and access patterns. Several causes make the selection of an appropriate set of aggregates a non-trivial task. First, there are no clear-cut transformation rules from a conceptual design to a document design, as efficient as the normalization process of relational databases [4]. Second, a wide range of alternative design options must often be considered (Fig. 2). When the number of entities and relationships is large, this may easily lead to a combinatorial explosion of alternative candidate schemas [4]. Third, a judicious modeling choice depends entirely on how we tend to manipulate data [1], and hence, must be *cost-driven* and influenced by the workload experienced by the system (*i.e. workload-aware*) [5]. Last and most importantly, it is commonly accepted that it is difficult, if not impossible, to find out a single set of aggregates suitable for all queries [4].

Replication is ubiquitous in NoSQL systems. In a previous work [6], we proposed a new replication scheme called *distorted replicas*. The main idea behind distorted replicas is to restructure replicated documents in different ways to better cope with the unavoidable heterogeneity of data access patterns. The idea of organizing replicated data in different ways was first introduced in [7] and applied to relational databases hosted on servers with mirrored disks. The idea was next applied to replicated blocks in distributed file systems [8]. We think that restructuring replicated data is much more of a central aspect for aggregate-oriented databases than it is for relational databases, since most applications will have to deal with queries that do not fit well with the aggregate structure.

The problem we tackle in this paper is as follows. Given a database with a replication factor of C (*i.e.* a database replicated C times) and an incoming query workload W, we have to determine C subsets of complete and disjoint aggregates that optimize W (*i.e.* that minimize the execution time of W). While there has been work in the area of document-oriented database design [4,9,10], we are not aware of any work that addresses the problem of aggregates selection and replication in an integrated manner. To deal with the logical and physical design challenges triggered by distorted replicas, we make the following key contributions: (i) we show how to identify interesting aggregates and how to assign them an *interestingness* value; (ii) we map the problem of aggregates selection to a *0-1 multiple knapsack problem* and solve it using a *branch and bound technique*; and (iii) we evaluate our approach on top of MongoDB using two real-world datasets: DBLP [3] and TPC-H [11]. The obtained results show that distorted replicas can execute queries up to tens of times faster than state-of-the-art approaches.

The remainder of this paper is organized as follows. Section 2 briefly reviews the main concepts related to document-stores. Section 3 discusses our workload-aware, cost-based algorithm for aggregates selection. Section 4 presents related works. Section 5

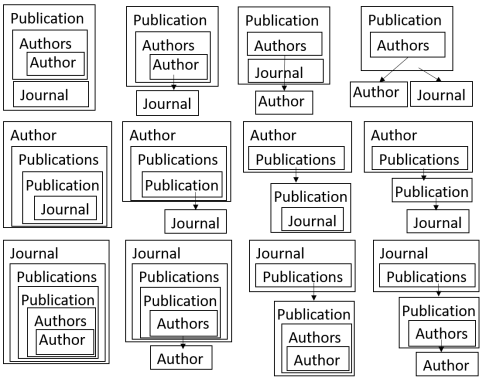

Fig. 2. Alternative schemas for the DBLP dataset; Nested rectangles represent embedded documents; Arrows represent references between aggregates.

gives an experimental study of distorted replicas performance. Section 6 discusses the main results and findings. Section 7 concludes the paper.

2. ANATOMY OF A DOCUMENT-STORE

There exists a wide range of academic and commercial document-stores, each with some features that may not exist in others. In the sequel, we use MongoDB as a representative of the feature set but also reference other document-store systems.

2.1 Document Modeling: Challenges and Considerations

2.1.1 Read-overhead

One of the most critical document-modeling choices is how to represent relationships between data: with references (*i.e. normalized data model*) or with embedded documents (*i.e. denormalized data model*). References represent relationships between data by including a link from one document to another, just as in the relational model. The normalized data model strives for a single copy of the data, minimizing redundancy and favoring consistency [4]. However, if related data is stored in separate servers, joins and writes may be prohibitively slow.

Embedded documents represent relationships by storing related objects in a single aggregate and hence, avoid cross-nodes joins and writes. Aggregates are useful in that they pack into one document, objects that are expected to be accessed together. However, there are many use cases where objects or fields need to be accessed individually. When a field needs to be accessed individually, not only that field's value is loaded in the memory hierarchy, but also all the data within the same aggregate. Loading large amounts of data irrelevant for a given query may seriously waste main memory, disk, and network bandwidths and increase the number of CPU cycles wasted in waiting for data loading [12]. The introduced *read-overhead* is one of the most important downsides of aggregate-oriented data models.

2.1.2 Aggregate roots

Objects in an aggregate are bound together by a root object, known as the *aggregate root* [1]. In most cases, there exist many root candidates. Grouping objects by one of the roots may help with some data interactions but is necessarily an obstacle for many others. As reported in [1], the entire aggregate orientation approach works well only when data access is aligned with aggregate roots. If data is accessed in a different way, the whole system performance may be substantially impacted. Consider the DBLP example of Fig. 1 and the relationship between authors and publications. Some queries will require to access authors whenever they access a publication; this fits in well with combining a publication with its authors into a single aggregate that can be stored and accessed as a unit. Other queries, however, will require to access the history of publications whenever they access an author. In such a case, it would be necessary to dig into every aggregate in the database. As aptly stated in [1], we can reduce this burden by building secondary indexes, but *we're still working against the aggregate structure*. Relational databases have an advantage here as they allow to slice and dice data in different ways for different queries.

In the following, an aggregate a is represented by a pair (r,E), where $a.E$ is the set of entities embedded within a and $a.r \in E$ is the root of a. An aggregate formed by a single entity is said to be *atomic*. To benefit from the efficiency of bitwise operations in our algorithms, $a.E$ is represented by a bitmap of $\mid \mathbb{E} \mid$ bits, where $\mathbb{E}$ is the set of modeled entities. The ith bit of the bitmap is set to 1 if the ith modeled entity is embedded within the aggregate, and to 0 otherwise.

2.2 Replication

Replication is the process of maintaining different replicas of the same data on different servers. The primary purpose of replication is to enhance availability and fault-tolerance by providing multiple paths to redundant data. Replication can also be used to increase: (i) I/O throughput by distributing requests across servers; and (ii) data locality by allowing a client application to access data from the closest server.

A set of servers maintaining replicas of the same data (sub-)set is called a *Replica Set* (or *RS*) in MongoDB. A replica set is composed of one master node, called *primary*, and a set of slave nodes, called *secondaries*. The primary node is the only member in a replica set that receives writes. When the primary receives a write request, it updates its data set and records the write in the operations Log (*i.e. opLog*). Secondary nodes periodically import the opLog and apply all changes to their local replicated collections in such a way that they reflect the master collections [13]. As in most NoSQL systems, replication in MongoDB is by default asynchronous: (i) there may exist a delay between the occurrence of an operation on the primary and its application on a secondary (*i.e. replication lag*); and (ii) the client application does not have to wait for the completion of a write on slaves.

2.3 Sharding

Sharding is similar to horizontal partitioning in RDBMSs. It consists in splitting a data set according to a given field, called the *shard key*. The resulting data subsets are called *chunks* and are hosted on multiple separate servers, called *shards*. Each shard is an *independent database* having its own subset of data stored on its own local disks. In MongoDB, each shard can be a complete replica set. A prominent concern in sharding

is to balance the load between shards. Typically, when a chunk grows beyond a given size, it is split causing an increase in the number of chunks held by the server. If the chunk distribution becomes uneven, some chunks are migrated from the shard that has the largest number of chunks to the one with the least number of chunks, until the cluster is rebalanced. A similar process occurs when a new shard is added to the cluster.

3. WORKLOAD-AWARE, COST-BASED AGGREGATES SELECTION

3.1 Overview

The main idea behind distorted replicas [6] is to restructure replicated data in different ways to better cope with the unavoidable heterogeneity of data access patterns. In our work, data restructuring is materialized by: (i) converting a reference between two aggregates into an embedded document and inversely, and (ii) reorganizing data according to different aggregate roots. Note here that restructuring aggregates according to new roots is not tedious as document-stores provide optimized operators allowing to promote an entity from "embedded" to "root" (*e.g.* `$replaceRoot` in MongoDB). We should also note that in our work data is only reorganized locally, *i.e.* within the same replica set (shard). Accordingly, even if references are used, we do not have to perform costly cross-nodes joins to reconstruct an aggregate.

We assume that we are given a database D with a replication factor C and a representative workload W, for which we need to recommend aggregates and their "distorted" replicas. The workload can be obtained when *migrating from RDBMS to NoSQL* as in [14, 15] or *by processing the Log of the database* as in [8, 16]. Our goal is to find C disjoint subsets of aggregates, such that the performance of W is optimized, subject to two constraints: (1) *Disjointness constraint*: aggregates within the same subset are disjoint, which means that within a replica set member, each modeled entity appears in at most one aggregate; (2) *Restorability constraint*: while not physically identical, replicas have to be logically identical. This means that all entities must appear within each replica set member.

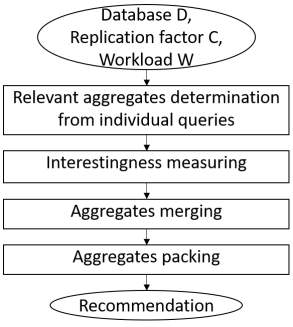

Fig. 3. Aggregates selection: key steps.

The key steps of our solution, pictured in Fig. 3, are in the spirit of S. Chaudhuri & V. Narasayya work on index, materialized views, and horizontal/vertical partitions selection in RDBMSs, summarized in the VLDB ten-year best paper award of S. Chaudhuri [16]. For simplicity of exposition, we retain the terminology used in [16] wherever applicable.

As illustrated in Fig. 2, an aggregate-oriented database designer is faced with a plethora of alternative design options: what entity is the owner of a relationship (*i.e.* what entity should embed a relationship), which relationships to denormalize, to which depth (*i.e.* subsequent relationships), *etc.*. Enumerating all possible combinations of embedding and referencing, becomes exponential as the number of entities and relationships increases [4]. To limit the space of the considered aggregates, we first restrict ourselves to those *relevant* for at least one query in the workload. Each relevant aggregate is then

assigned with a value quantifying its *interestingness*.

An aggregate suitable for one query, may substantially degrade the performance of another query. The goal of the "relevant aggregates merging" step is hence to find additional aggregates, that although are not optimal for any individual query, are useful for multiple queries, and therefore could be optimal for the workload. Given the set of interesting aggregates and a replica set of C members, the "aggregates selection" step aims to select C subsets of complete and disjoint aggregates so that the total interestingness of the replica set is a maximum.

As in [5,8,16], we assume that there is a function $Cost(q,a)$ that returns the optimizer estimated cost of a query q when q is answered using an aggregate a. As stated in [16], many database systems support the necessary interfaces to answer such "what-if" questions. In the absence of such interfaces, we could follow the same approach as [5, 8] and estimate $Cost(q,a)$ by the footprint of q, *i.e.* by the total number of bytes read (*i.e.* consumed) by q. The footprint of q corresponds to: $Size(a) \times Cardinality(a) \times Selectivity(q)$, where $Size(a)$ is the estimated size of a, $Cardinality(a)$ the estimated number of a instances, and $Selectivity(q)$ the percentage of instances that q selects (with 0 meaning no instances and 1 meaning all instances). If no index is used, a full scan is necessary and $Selectivity(q)$ is dropped from the formula. Without any loss of generality, one can supply other $Cost(q,a)$ estimations to our algorithm.

3.2 Determining Relevant Aggregates from Individual Queries

In the following, the term *rootable entities* is used to denote entities in a query that can potentially be good aggregate roots. Rootable entities form the basis for "determining relevant aggregates from individual queries", the first step of our approach (Fig. 3). We define rootable entities as follows. Let $q.E$ be the set of modeled entities referenced in a query $q \in W$. Intuitively, an entity $e \in q.E$ is *rootable* for q, if it is potentially useful to group the entities referenced in q by e (*i.e.* to create an aggregate rooted at e that embeds all of the entities referenced in q). We consider that an entity $e \in q.E$ is rootable for q if there exists a field f of e such that f appears in the `Group By` clause, the `Order By` clause, or in a filter predicate of q. In the absence of at least one entity satisfying one of the above conditions, each $e \in q.E$ is considered as a rootable entity.

Intuitively, an aggregate a is *relevant* for a query q if q is answerable using a, hence, if all of $q.E$'s entities are embedded within a ($q.E \subseteq a.E$). Considering all relevant aggregates for a query is not scalable since, in principle, we would have to consider any aggregate a such that $q.E \subseteq a.E$. To prune the space of relevant aggregates, we only consider aggregates whose roots are rootable entities and approach the task of "relevant aggregates selection" in two steps. From each query $q_i \in W$, we first derive a preliminary set A_i of aggregates relevant for q_i. Each of those aggregates: (i) is rooted at a rootable entity for q_i; and (*ii*) contains exactly all of the entities referenced in q_i (and nothing else). For each $a_j \in A_i$, we next check its relevance for the remaining workload queries. The output of this stage is a relevance matrix $R(q_i,a_j)$, indicating whether or not an aggregate a_j is relevant for a query q_i, *i.e.* $R(q_i,a_j) = 1$ if a_j is relevant for q_i, and 0 otherwise.

Example. Let's consider the following self-explanatory SQL queries.
q_1: *"select a.mail from author a where a.name=.."*
q_2: *"select a .id, count(*) from author a, publication p where p.year=.. group by a.id"*.

a_1=*(author,*{*author*}*)* is a relevant aggregate for q_1. a_2=*(author,*{*author, publication*}*)* and a_3=*(publication,*{*author, publication*}*)* are relevant aggregates for q_2.
As $a_1.r = a_2.r$ and $a_1.E \subseteq a_2.E$, we conclude that a_2 is also relevant for q_1. Accordingly, $R(q_1,a_2)$ is set to 1. In contrast to a_2, a_3 is not relevant for q_1, since $a_3.r \neq a_1.r$. $R(q_1,a_3)$ does not change and remains equal to 0, as shown in the relevance matrix opposite.

	q_1	q_2
a_1	1	0
a_2	1	1
a_3	0	1

△

3.3 Measuring Interestingness of Relevant Aggregates

Our next goal is to define a metric that captures the relative *interestingness* of each relevant aggregate. Such a metric is essential to rank aggregates and pick the most valuable ones in the final replica set. Intuitively, an aggregate is interesting for a workload W, if it speeds up a significant fraction of W's queries, *i.e.* allows to significantly reduce the total cost of W. The relevance matrix is useful for indicating which aggregates are relevant for which queries. However, it cannot be used by its own to determine interestingness as it does not take into account neither the relative importance (*i.e.* footprints) of queries nor the read overhead introduced by answering a query q using an aggregate that embeds entities useless for q (not referenced in q).

Let A be the set of all relevant aggregates, $A_i \subseteq A$ the set of aggregates relevant for a query q_i ($\forall\, a_j \in A$, $R(q_i,a_j) = 1$), $Opt(W)$ the optimal cost of W, and $Opt(q_i)$ the optimal cost of q_i. $Opt(q_i)$ is defined as the lowest achievable cost to answer q_i: $Opt(q_i) = Min_{a_j \in A_i}(Cost(q_i,a_j))$. $Opt(W)$ is the lowest achievable cost of W: $Opt(W) = \sum_{1}^{|W|} Opt(q_i)$. We define $Int(a) \in [0,1]$, the *interestingness* of an aggregate a for a workload W, as the fraction of the cost of all queries in W for which a is relevant. Formally, $Int(a) \rightarrow [0,1]$ is defined as follows: $Int(a) = \frac{\sum_{1}^{|W|} R(q_i,a) \times \frac{Opt(q_i)^2}{Cost(q_i,a)}}{Opt(W)}$.

$Int(a)$ is normalized by the total workload cost to make it comparable. This can be used as well if it is necessary to prune aggregates by discarding from further consideration those whose interestingness is below a predefined threshold.

Example. Consider for simplicity a workload consisting of one query q and 3 relevant aggregates a_1, a_2, a_3, such that $Cost(q,a_1)$ = 10 units, $Cost(q,a_2)$ = 15 units and $Cost(q,a_3)$ = 20 units. The relative interestingness is: $Int(a_1) = 1$, $Int(a_2) = 0.67$ and $Int(a_3) = 0.5$. △

Example. Assume that W consists of 2 queries q_1 and q_2 and that we have 4 relevant aggregates a_1, a_2, a_3, and a_4. Each cell in the following table gives the cost of answering a query q_i using an aggregate a_j (with "-" meaning that a_j is not relevant for q_i).
As illustrated in the table opposite, a_1 is optimal for q_1 and a_2 is optimal for the more expensive query q_2 (having higher optimal cost). a_3 and a_4 allow to answer q_1 and q_2 but introduce a read overhead. The interestingness of each aggregate is: $Int(a_1) = 0.33$, $Int(a_2) = 0.67$, $Int(a_3) = 0.76$ and $Int(a_4) = 0.5$.

	q_1	q_2
a_1	10	-
a_2	-	20
a_3	**15**	**25**
a_4	20	40

Intuitively, a_2 is more interesting than a_1 for the given workload, as it is optimal for a query more expensive than the query for which a_1 is optimal. a_3 is more interesting than a_4 as it allows to answer q_1 and q_2 with a lower read overhead. a_3 is more interesting than a_2 as it allows to answer q_1 without introducing a "high" additional cost for q_2. △

3.4 Merging Pairs of Interesting Aggregates

The *disjointness constraint* states that non-disjoint aggregates cannot be stored within the same replica set member. This is essential to avoid data redundancy and, hence, data inconsistency. Due to the disjointness constraint, a large part of relevant aggregates derived from individual queries will be rejected and we can end up with sub-optimal recommendations for the workload. The intuition behind "aggregates merging" is that merging two mutually exclusive aggregates, called the *parent aggregates*, in one sub-optimal aggregate, called the *merged aggregate*, is in some cases better than retaining one parent and rejecting the other. As the merged aggregate and its parents are mutually exclusive, the merged aggregate: (i) should be usable (*i.e.* relevant) in answering all queries where each of its parents was used; and (ii) the cost of answering queries using it should not be "much higher" than the cost of answering queries using one of its parents.

In our work, we only merge aggregates that meet the two following criteria. The parent aggregates must have: (i) the same root; and (ii) exactly one non-common entity with the merged aggregate. The first condition is necessary to ensure that the merged aggregate is relevant for all queries for which one of its parents is relevant. The second condition ensures that the cost of answering these queries using the merged aggregate is not "much higher" than the cost of answering them using its parents. The merged aggregate is retained only when its interestingness is greater than the lowest interestingness of its parents.

Example. Let $a_3 = (r, \{r,s,t\})$ be a merged aggregate and $a_1 = (r, \{r,s\})$, $a_2 = (r, \{r,t\})$ its parent aggregates. Assume that the cost matrix is as follows.

	q_1	q_2
a_1	10	-
a_2	-	20
a_3	20	40

The interestingness of a_1, a_2 and a_3, is respectively $Int(a_1) = \frac{1}{3}$, $Int(a_2) = \frac{2}{3}$ and $Int(a_3) = \frac{1}{2}$. a_3 is therefore retained.
Suppose now that $Cost(q_1, a_3) = 50$ and $Cost(q_2, a_3) = 100$. In such case, $Int(a3) = \frac{1}{5}$ and a_3 is not retained as retaining a_2 is a better choice (the gain achieved is lower than the loss caused by the read-overhead). △

3.5 Interesting Aggregates Selection as a 0-1 MKP

Given the set I of interesting aggregates and C replica-set members ($C \ll |I|$), our goal is to select C disjoint subsets of I, so that: (i) the total interestingness of the selected aggregates is a maximum; and (ii) the disjointness and the recoverability constraints are met. The aggregates selection problem can be likened to a 0-1 multiple knapsack problem (MKP) which is known to be NP-hard. More precisely, given C knapsacks (replica set members) and N items (interesting aggregates), we have to find binary variables x_{ij}, $i \in \{1..C\}$, $j \in \{1..N\}$, having the following meaning: $x_{ij} = 1$ if aggregate j is assigned to member i, and 0 otherwise. Formally, the problem is stated as follows. Let B be a bitmap with $\mid \mathbb{E} \mid$ bits all set to 1:

```
   Input  : items, knapsacks, level, nodeID, bestNode, maxProfit
   Output: Id of the decision-tree node that corresponds to the optimal solution
 1 if j < items.length() − 1 then
 2     B&B(items, knapsacks, level+1, nodeID + ".0", bestNode, maxProfit)
 3 for i = 0; i < knapsacks.length(); i++ do
 4     if Hamming_weight(knapsacks[i].bitmap ∧ items[level].bitmap) == 0 then
 5         knapsacks[i].profit += items[level].profit
 6         knapsacks[i].bitmap = knapsacks[i].bitmap ∨ items[level].bitmap
 7         if level < items.length() − 1 then
 8             B&B (items, knapsacks, level+1, nodeID + "." + str(i+1), bestNode,
                maxProfit)
 9         else if knapsacks[i].profit > maxProfit then
10             maxProfit = knapsacks[i].profit
11             bestNode = nodeID
```

Fig. 4. A Branch&Bound algorithm for aggregates packing.

$$\text{maximize} \sum_{i=1}^{C} \sum_{j=1}^{N} Int(a_j) x_{ij} \qquad \text{subject to,}$$

$$x_{ij} \in \{0,1\}, \qquad i \in \{1,\ldots,C\},\ j \in \{1,\ldots,N\} \quad (1)$$

$$\sum_{i=1}^{N} x_{ij} = 1, \qquad j \in \{1,\ldots,N\} \quad (2)$$

$$\sum_{j=1}^{N} w_j x_{ij} \leq B \qquad i \in \{1,\ldots,C\} \quad (3)$$

Constraint (1) is self-explanatory. Constraint (2) indicates that a given aggregate cannot be assigned to more than one replica set member. Constraint (3) (disjointness constraint) *substitutes the classic capacity constraint of the knapsack problem* and indicates that a given member cannot hold two non-disjoint aggregates. When no confusion is possible, the terms item / aggregate, knapsack / RS member, and profit / interestingness are used interchangeably in the sequel.

To solve the aggregates selection problem, we opt for a depth-first branch-and-bound approach. The pseudo-code of our algorithm is shown in Fig. 4. In this algorithm, the decision tree's successive levels are built by selecting a branching aggregate and assigning it to each knapsack in turn. The branching aggregate is first assigned to a dummy knapsack (lines 1–2), implying its exclusion from the current solution. The aggregate is then assigned to each knapsack that satisfies constraint (2) (lines 3–8). If constraint (2) is satisfied (*i.e.* the conjunction of the knapsack bitmap and the item bitmap is 0), B&B is called recursively to continue exploring the corresponding sub-tree. In the other case, the branch is ignored. This is illustrated in the example of Fig. 5, where knapsack 0 is the dummy knapsack and aggregates a_1 and a_2 are not disjoint. The branches that assign a_1 and a_2 to the same "non-dummy" knapsack are discarded from further consideration (node 1.1 and node 2.2).

In case we have explored all items in a given path, we check if we have a greater

profit than before and update the optimal solution (lines 9-11). Once the set of aggregates assigned to each member is determined, we have to check the recoverability constraint. Indeed, if the disjunction of the bitmaps of aggregates assigned to a knapsack K_i is a bitmap different than B, this means that one or more modeled entity is not represented in K_i. In such a case, an atomic aggregate for each of those entities is added to the member.

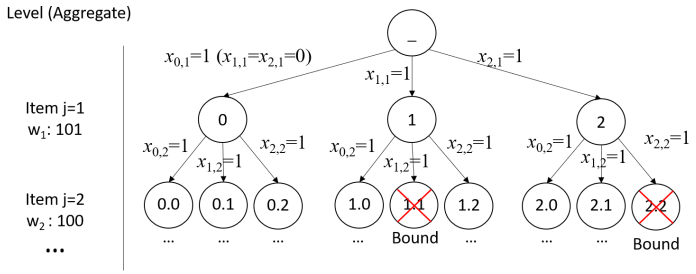

Fig. 5. Item 1 and item 2 are not disjoint. The branches that assign them to the same knapsack are discarded (Branch&Bound).

4. RELATED WORK

4.1 Trojan Data Layouts

[8] proposes Trojan Layouts, a data layout inspired by PAX (Partition Attributes Across) and intended to improve data access times in Hadoop Distributed File System (HDFS). Given a relation R with arity n, PAX partitions each block into n mini-blocks. The ith mini-block stores all the values of the ith attribute of R. Trojan Layouts split an HDFS block into $m \leq n$ mini-blocks and store in each mini-block the values of k ($1 \leq k \leq n$) attributes (*i.e.* vertical partitioning inside each chunk). Trojan Layouts provide a high degree of spatial locality when the values of the k grouped attributes are sequentially accessed and avoid to read $(n-k)$ irrelevant attributes for a given query. To better handle a mix of queries with different access patterns, [8] also proposes to group attributes differently in each HDFS block replica according to the query workload.

Trojan Layouts reorganize a modeled entity's attributes at the block level, while distorted replicas reorganize entities at the database level. Trojan Layouts are then only useful for queries touching a limited subset of an entity's attributes, whereas distorted replicas are useful for queries touching more than one entity.

4.2 Secondary Indexes and the Divergent Physical Design

Numerous techniques have been proposed to build secondary indexes on NoSQL databases. In [17] the authors propose a novel tuning paradigm for replicated databases, called *divergent designs*. Given a replicated database, a divergent design indexes the same data differently in each replica, and hence, specializes replicas for different subsets of the workload [17]. With this design, each query is routed to the replica that can evaluate it most efficiently. The idea of divergent design was further developed in [18], where the authors proposed RITA, an index-tuning advisor for replicated databases. RITA allows to: (i) generate fault-tolerant divergent designs; and (ii) spread the load evenly over replicas.

With secondary indexes, some of the latency would be hidden, but performance would only be sub-optimal since we're still "working against the aggregate structure" as aptly stated in [1] (a further discussion on secondary indexes is given in Section 6).

4.3 Schema Modeling

Despite the wide diffusion of document-oriented DBs, little work has been devoted to their modeling [15]. The work of [9] introduces an aggregates-based logical data model, called NoAM (*NoSQL Abstract Model*), and demonstrates how data modeled in NoAM can be implemented in different NoSQL types. The NoAM modeling approach consists of four steps: (i) aggregate design: the classes of aggregated objects needed for an application are identified (conducted by use cases and performance requirements); (ii) aggregate partitioning: aggregates are partitioned into smaller data elements; (iii) high-level NoSQL DB design: aggregates are mapped to the NoAM model according to the identified partitions; and (iv) implementation: the NoAM schema is converted to the schema of the target NoSQL DB type. Although [9] admits that "*aggregate design is mainly driven by data access operations*", it does not provide a practical workload-aware, cost-based approach for identifying aggregates.

The work of [4] explores different schema design tactics and provides general guidelines for modeling document-oriented DBs. Notably, [4] advocates a workload-aware design consisting of two phases: (i) leveraging common heuristics to generate a finite number of candidate schemas (*i.e. candidate generation*); and (ii) ranking these candidate schemas using cost functions (*i.e. candidate ranking*). [4] states that there are two approaches to candidate generation: *top-down* and *bottom-up*. The bottom-up approach starts with a normalized schema and optimizes each query by adding denormalized structures. The top-down approach starts with a set of globally optimal aggregates that answers each query with a single look-up. Our work follows the broad lines of [4] and can be seen as an implementation of the top-down approach. In our work, however, we go further than [4] (and [9]) and propose a set of practical algorithms and cost functions for the identification, evaluation, and selection of aggregates.

4.4 Database Migration from RDBMS to Document-Oriented NoSQL

As mentioned previously, join operations are rarely supported in document-stores and are often processed at the application layer, in a much more expensive way than RDBMS. Unsurprisingly, most of the existing work on database migration from RDBMS to document-oriented NoSQL [14, 15] propose variants of the denormalized form and try to best balance the trade-offs between the normalized form and the denormalized form. We distinguish two types of denormalization: *table-level denormalization* and *column-level denormalization*. For table-level denormalization, [14] proposes a Breadth-First Search algorithm to find a path from a root table (*i.e.* root entity) to other related tables. This path is used for creating a document template for the root table (by recursively embedding, in a Breadth-First Search fashion, related tables). To reduce the number of embedded entities, the approach of [14], referred to as BFS in the sequel, exploits each link between entities not more than once (an example is given in Fig. 8 (b), where the entity *Region* is embedded only once). As noticed in [19], BFS adds too many weakly related tables in the root document.

The most recent work on denormalization [15] proposed CLDA (*Column-Level Denormalization with Atomicity*), a column-level based denormalization. Rather than embedding entire entities inside aggregates, CLDA preserves all original entities in separate aggregates and duplicates only those columns accessed in non-primary-foreign-key-join predicates (*e.g.* filter predicates). By doing so, CLDA aims at avoiding join operations while minimizing the read-overhead. CLDA can be considered as an implementation of the bottom-up approach discussed in [4]. An important downside of CLDA is that it introduces data redundancy, which is often a source of inconsistency. In contrast with our approach, CLDA is not cost-driven, *i.e.* does not take into account neither the relative importance of queries nor their costs. Furthermore, CLDA systematically duplicates columns without considering the cases where the gain achieved through duplication is lower than the loss caused by the read-overhead. An experimental evaluation of BFS [14], CLDA [15], and distorted replicas is given in Section 5.

5. EXPERIMENTAL EVALUATION

Distorted replicas were implemented on top of MongoDB release 3.6. Experiments were performed on dedicated dual-core i5-3230M systems, running Ubuntu 16.04.1 LTS. Each core offers a base speed of 2.6 GHz and the two cores can handle up to four simultaneous threads. These computers feature 16 GB main memory (DDR3-1600MHz), 128 kB L1 cache, 512 kB L2 cache, and 3 MB L3 cache. The hard disk is a Serial-ATA/600 having a rotational speed of 7200 rpm. MongoDB was run using its default settings and no special tuning was done. All queries were implemented using the MongoDB Aggregation Pipeline. For each query, we report the average execution time of three consecutive runs. We ran our experiments with two main objectives in mind: (i) to show that distorted replicas allow improving data access performance significantly (Subsection 5.1), and (ii) to evaluate the effectiveness of our aggregates selection algorithm (Subsection 5.2).

5.1 Distorted Replicas Effectiveness

We compared distorted replicas with two state-of-the-art approaches: BFS [14], a table-level denormalization method, and CLDA [15], a column-level denormalization method. BFS, CLDA, and distorted replicas were evaluated using two complementary real-world datasets: DBLP [3], a relatively simple workload, and TPC-H [11], a more complex workload. In this paper, we are focusing on reorganizing data within the same replica set and are less concerned with sharding[1]. We then assume that we are given a MongoDB cluster consisting of one replica set or a MongoDB cluster consisting of several shards, each deployed as a replica set. Our aim is to replicate the database hosted at a replica set in different ways and to evaluate the gains in execution time.

5.1.1 DBLP

The DBLP Computer Science Bibliography dataset [3] contains bibliographic information on scientific publications. All the DBLP records are distributed in one big XML file. Each record is associated with a set of fields representing bibliographic data relevant

[1]The study of the effect of data movements between shards (between replica-sets) is part of our ongoing work.

with respect to its type and has an *id* field that uniquely identifies it. We developed a DBLP parser in Java following the recommendations of [3]. Currently, our parser only extracts "article" and "inproceeding" records. The extracted records were inserted in the same collection. The resulting MongoDB collection contains $\approx$ 3.1 million publication documents, embedding $\approx$ 1.6 million distinct authors, and $\approx$ 9.5 thousand distinct journals/conferences. The average document size is 538 bytes, and the total collection size is $\approx$ 1.5 GB. The considered workload is intentionally basic, so that the fundamental properties of CLDA, BFS, and distorted replicas can be better illustrated and highlighted. The workload consists of the following queries.

- q_1: *"Find the authors of a given publication"*. In the parsed DBLP dataset, the average number of authors per publication is $\approx$ 2.85. We randomly selected 3 publications co-written by 3 authors and reported the average execution time.
- q_2: *"Find the publication titles of a given author"*. The average number of publications per author is $\approx$ 5.46. We randomly selected 3 authors with 6 publications each and measured the average execution time. As authors may have variations in their first names (*e.g.* "Mike Stonebraker", "Michael Stonebraker", *etc.*), we used a regular expression to find out publications (*e.g. author._id:* {*/Stonebraker$/*}).
- q_3: *"Find the number of publications per year"*.

The considered BFS aggregate, depicted in Fig. 6 (a), is rooted at *Publication* and embeds *Author* and *Journal*. As shown in Fig. 7, BFS is slightly slower than CLDA and distorted replicas for q_1. This is due to the fact that BFS embeds *Journal* within the aggregate rooted at *Publication*, whereas CLDA and distorted replicas do not. Fig. 7 also shows that BFS outperforms CLDA for q_3 but turns out to be very slow when used to answer q_2. The poor performance of BFS for q_2 is caused by the necessity to visit each *Publication* document and, for each document, to iterate over its *Authors* array (analogously, if the BFS aggregate were rooted at *Author*, it would be slightly slower than CLDA and distorted replicas for q_2, but very slow for q_1 and q_3).

CLDA is optimal for q_1 and q_2. However, in the absence of a query involving *Publication* and containing a filter predicate on *Journal.year*, the relationship between *Publication* and *Journal* is not denormalized. Answering q_3 using CLDA requires therefore a costly join operation (Fig. 7).

Fig. 6 (b) illustrates the set of aggregates selected by our approach (for $C = 3$). As shown in Fig. 7, distorted replicas are optimal for q_1 and q_3 and only slightly slower than CLDA for q_2 (as distorted replicas embed the entire *Publication* entity within the aggregate rooted at *Author*, whereas CLDA only embeds *Publication._id* and *Publication.title*). Overall, distorted replicas are, respectively, $\approx$ 2.8 and $\approx$ 1.28 faster than BFS and CLDA for the considered workload (an improvement factor of 2 for a query q or a workload W means that q/W is executed 2 times faster).

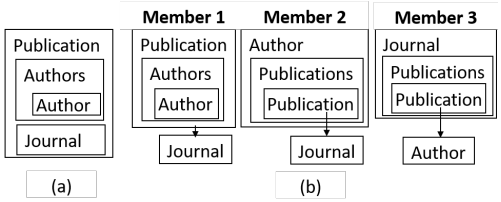

Fig. 6. DBLP dataset; (a) BFS DB; (b) Distorted replicas ($C = 3$).

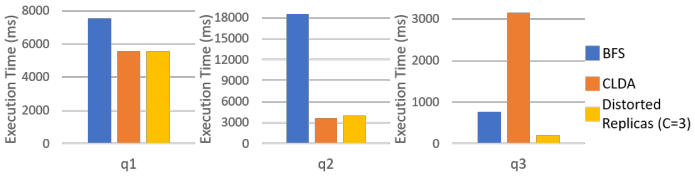

Fig. 7. Query performance (DBLP).

5.1.2 TPC-H

We generated TPC-H data using the TPC-H DBGEN data generator tool with a scale factor of 10 and developed a Java loader module to import TPC-H data in MongoDB.

We considered 10 representative TPC-H queries: q_1, q_3, q_5, q_6, q_{10}, q_{11}, q_{14}, q_{15}, q_{17}, q_{19}. The outputs of our algorithm steps are synthesized in Table 1. The BFS approach could produce different denormalized schemas, depending on the order of edge visits [14, 15]. We adopted the schema described in [15] and illustrated in Fig. 8 (b). As shown in Fig. 8 (b), the BFS DB is very close to a fully denormalized DB having the same root. The only difference is that *Region* is embedded only once.

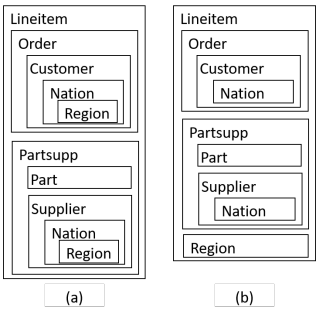

Fig. 8. TPC-H dataset; (a) Fully denormalized DB; (b) BFS DB.

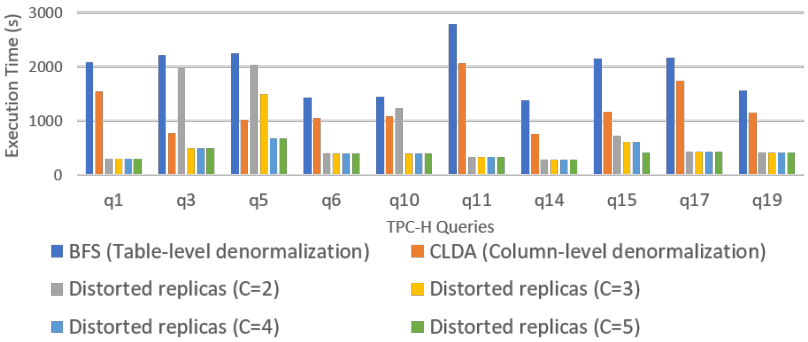

Fig. 9. Query performance (TPC-H).

Fig. 9 depicts the performance achieved by BFS, CLDA and distorted replicas, in terms of query execution time. Fig. 10 reports the improvements achieved by distorted replicas over BFS and CLDA as a function of the replication factor C. As shown in Fig. 9 and Fig. 10, BFS is by far outperformed by CLDA and distorted replicas (resp., ≈ 1.51 and up to ≈ 4.97 times faster). To understand the behavior of distorted replicas and CLDA, let's consider the atomic aggregate $\{Lineitem, (Lineitem)\}$, which is optimal for queries q_1 and q_6 and contributes to optimizing q_{15} and q_{17}. The atomic aggregate $\{Lineitem, (Lineitem)\}$ is selected by our knapsack algorithm $\forall\, C \geq 2$. Due to queries

such as q_5, q_{14}, and q_{19}, the relationships between *Lineitem*, *Supplier*, *Part*, and *Nation* are partly denormalized in CLDA (*i.e.* some columns from *Supplier*, *Part*, and *Nation* are embedded within *Lineitem*). The embedded columns are useless for queries q_1, q_6, q_{15} and q_{17}, and introduce a substantial read-overhead. Furthermore, CLDA nests *Lineitem* into *Order* as an array of sub-documents to better support atomicity [15]. Answering q_1, q_6, q_{15}, and q_{17} using CLDA, requires therefore an expensive *unwind* operation (to flatten the array of sub-documents).

As expected and shown in Fig. 9, distorted replicas outperform CLDA for these queries. Fig. 9 also shows that when C is low, CLDA performs better than distorted replicas for some queries (*e.g.* q_5 and q_{15}). This is due to the disjointness constraint, which does not allow to select all optimal aggregates. As shown in Fig. 9, when C is increased, more optimal aggregates are selected and more queries are optimized.

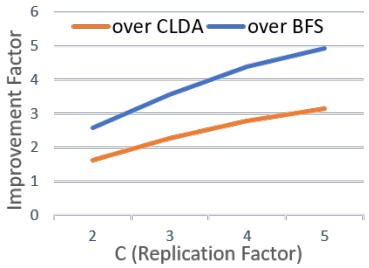

Fig. 10. Improvement factor as a function of the Replication Factor.

5.2 Selection Algorithm Performance

Now we focus on the effectiveness of our algorithm of aggregates selection.

5.2.1 Number of Iterations.

First of all, we show the effect of adding the disjointness constraint to our knapsack formulation. Recall that the "disjointness constraint" substitutes the classic capacity constraint of the knapsack problem and is used to prune the search space. Table 2 compares the number of iterations with and without the disjointness constraint as a function of the replication factor (*i.e.* number of knapsacks). The number of iterations without the disjointness constraint corresponds to the total number of nodes in the decision tree. As shown in Table 2, the disjointness constraint substantially reduces the number of iterations in our algorithm: the fraction of visited nodes for $C = 5$ is $\approx$ 1.17803E-07.

Table 1. Aggregates selection algorithm applied to TPC-H queries.

# of queries	# of relevant aggregates	# of merged pairs	# of selected aggregates (C=2)	# of selected aggregates (C=3)	# of selected aggregates (C=4)	# of selected aggregates (C=5)
10	18	3	3	4	5	6

Table 2. Knapsack iterations as a function of the replication factor C (N=21 items).

	C=2	*C=3*	*C=4*	*C=5*
With Disjointness Constraint	47 600	2 252 212	84 579 458	2 584 233 662
Without Disjointness Constraint	10 460 353 203	4.39805E+12	4.76837E+14	2.1937E+16

Table 3. Execution time as a function of the replication factor C (N=21 items).

C=2	*C=3*	*C=4*	*C=5*
794 ms	8 s	302 s	172 m

5.2.2 Execution time

Table 3 shows the time taken to create aggregate groups, as a function of the replication factor C. The execution time of our algorithm is highly dominated by the resolution of the MKP problem using the branch-and-bound algorithm. For 21 interesting aggregates and 5 replicas, the time taken to group aggregates is around 172 minutes. This is acceptable, given that grouping is an offline process. Recall that, as suggested in Subsection 3.3, the execution time can be reduced by discarding aggregates whose interestingness is below a predefined threshold.

6. DISCUSSION

Like materialized views and secondary indexes, distorted replicas are intended to improve query performance for workloads of common and repeated query patterns. In this work, we showed that distorted replicas substantially reduce query execution time. A salient feature of our work is that it maps the problem of aggregates packing to a 0-1 Multiple Knapsack problem and solves it using a branch and bound technique.

Typically, database systems gather statistics on search queries and provide tools for diagnosing database performance. In the same way as materialized views and secondary indexes, database admins can generate distorted replicas (and eventually delete less useful ones) when they witness slow execution times due to new query patterns. Distorted replicas can also be generated when a database is migrated from RDBMS to NoSQL or at design time (in the latter case, the DB designer will have to estimate relationship cardinalities and document sizes). As materialized views, distorted replicas restructure documents to provide new ways for exploring data. The main difference is that distorted replicas take advantage of the already-existing replication to generate restructured data without any additional refresh cost, while materialized views introduce a significant write overhead. It is worth noticing that if materialized views were enabled to regenerate base data in a two-way replication scenario, they would be an interesting tool to implement distorted replicas (such a scenario is permitted in some RDBMSs such as Oracle).

Compared to distorted replicas, secondary indexes only allow sub-optimal performance. Consider for example the DBLP dataset where data is organized by *Publication* and a query such as "Find Stonebraker's publications". A secondary index on author identifiers allows only retrieving the subset of documents with an author named Stonebraker. In contrast, with a distorted replica where data is organized by *Author*, only one document embedding all Stonebraker's publications would be retrieved. Secondary indexes are also not helpful for map/reduce jobs and queries requiring a full scan or involving regular expressions. With the DBLP dataset and the MongoDB settings of Section 5, a query looking for the publications of any author having "Stonebraker" as last name (regex: *author._id:* {*/Stonebraker$/*})), takes on average 49980 ms with a secondary index, 18505 ms without an index and only 3849 ms if data is organized by *Author*.

7. CONCLUSION

Aggregates are useful in that they pack into one document, data that is expected to be accessed together. Aggregates are essential to data processing at the level of large-scale clusters, but severely limit the ways data can be efficiently explored and processed. This paper deepened the idea of distorted replicas, a replication scheme that restructures replicated data in different ways to improve data access times. In particular, we showed how to: (i) identify relevant aggregates; (ii) assign interestingness values to relevant aggregates; (iii) merge pairs of interesting aggregates; and (iv) pack interesting aggregates in such a way that the total interestingness of a replica set is maximized. We also implemented our ideas on top of MongoDB and evaluated distorted replicas using two real-world datasets. Experimental results show that distorted replicas substantially reduce query execution time: up to tens of times faster than state-of-art methods.

As a part of our future work, we intend to define a cost model to help query optimizers determine the replica to which a query should be directed. Another interesting point to consider is a closer study of dynamic migration of data between shards and its impact on distorted replicas and on load balancing.

ACKNOWLEDGMENT

We would like to thank Pr. Ouajdi Korbaa for many helpful discussions and reviews that improved this paper.

REFERENCES

1. P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, Addison-Wesley Professional, NJ, 2012.
2. A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino, "Persisting big-data: The nosql landscape," *Information Systems*, Vol. 63, 2017, pp. 1-23.
3. M. Ley, "DBLP – some lessons learned," in *Proceedings of VLDB Endowment*, Vol. 2, 2009, pp. 1493-1500.
4. V. Reniers, D. V. Landuyt, A. Rafique, and W. Joosen, "Schema design support for semi-structured data: Finding the sweet spot between NF and De-NF," in *Proceedings of IEEE International Conference on Big Data*, 2017, pp. 2921-2930.
5. C. de Lima and R. dos Santos Mello, "A workload-driven logical design approach for nosql document databases," in *Proceedings of the 17th International Conference on Information Integration and Web-Based Apps & Services*, 2015, pp. 1-10.
6. K. Jouini, "Distorted replicas: Intelligent replication schemes to boost I/O throughput in document-stores," in *Proceedings of IEEE/ACS 14th International Conference on Computer Systems and Applications*, 2017, pp. 25-32.
7. R. Ramamurthy, D. J. DeWitt, and Q. Su, "A case for fractured mirrors," *The VLDB Journal*, Vol. 12, 2003, pp. 89-101.
8. A. Jindal, J. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: right shoes for a running elephant," in *Proceedings of ACM Symposium on Cloud Computing*, 2011, p. 21.

9. P. Atzeni, F. Bugiotti, L. Cabibbo, and R. Torlone, "Data modeling in the NoSQL world," *Computer Standards & Interfaces*, Vol. 6, 2020, pp. 103-149.
10. V. Varga, C. Sacarea, and A. É. Molnár, "Conceptual graphs based modeling of semi-structured data," in *Proceedings of the 23rd International Conference on Conceptual Structures*, LNCS 10872, 2018, pp. 167-175.
11. "The TPC-H benchmark," http://www.tpc.org/tpch, 2020.
12. K. Jouini, G. Jomier, and P. Kabore, "Read-optimized, cache-conscious, page layouts for temporal relational data," in *Proceedings of the 19th International Conference on Database and Expert Systems Applications*, LNCS 5181, 2008, pp. 581-595.
13. "MongoDB," http://www.mongodb.com/guides/, 2019.
14. G. Karnitis and G. Arnicans, "Migration of relational database to document-oriented database: Structure denormalization and data transformation," in *Proceedings of the 7th International Conference on Computational Intelligence, Communication Systems and Networks*, 2015, pp. 113-118.
15. J. Yoo, K. Lee, and Y. Jeon, "Migration from RDBMS to NoSQL using column-level denormalization and atomic aggregates," *Journal of Information Science and Engineering*, Vol. 34, 2018, pp. 243-259.
16. S. Chaudhuri and V. R. Narasayya, "Self-tuning database systems: A decade of progress," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, 2007, pp. 3-14.
17. M. P. Consens, K. Ioannidou, J. LeFevre, and N. Polyzotis, "Divergent physical design tuning for replicated databases," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2012, pp. 49-60.
18. Q. T. Tran, I. Jimenez, R. Wang, N. Polyzotis, and A. Ailamaki, "RITA: an index-tuning advisor for replicated databases," in *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, 2015, pp. 22:1-22:12.
19. B. Namdeo and U. Suman, "Performance analysis of schema design approaches for migration from RDBMS to NoSQL databases," in *Advances in Data and Information Sciences*, 2020, pp. 413-424.

Khaled Jouini received the Ph.D. degree in Computer Science from Paris-Dauphine University, France. He was a research staff member at Telecom ParisTech, France. Since 2011, he has been with Sousse University, Tunisia, where he is currently an Associate Professor. His research interests include non-volatile memory, database systems, and large-scale data management and mining.