Distorted Replicas: Intelligent replication schemes to boost I/O throughput in document-stores

Khaled Jouini
MARS Research Lab LR17ES05
ISITCom, University of Sousse, Tunisia
Email: j.khaled@gmail.com

***Abstract*—NoSQL databases commonly use an aggregate data model, where data that is expected to be accessed together is packed in a single clump and stored in a single node [1]. This aggregate-orientation is essential to running on a cluster as it avoids cross-nodes joins and writes. An important downside of the aggregate data model is that it severely limits the ways data can be efficiently explored and processed, especially as NoSQL systems are being increasingly used by complex data-driven applications, mixing heterogeneous data access patterns.**

Replication is ubiquitous in NoSQL systems. In this work we propose a new replication scheme termed *distorted replicas*. Rather than being physically identical, distorted replicas are logically identical : they restructure replicated data in different ways, but keep the fundamental property of being constructible from one another. By doing so, distorted replicas provide new ways for exploring data, while still ensuring high availability. Experiments conducted in this paper show that even basic distortion schemes allow substantial performance improvements.

***Index Terms*—Aggregate Data Model, Document-Stores, Replication, Physical Design**

I. 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]. To ensure high availability, NoSQL systems maintain multiple copies of data on different servers. Most of existing systems implement an *asynchronous replication* which compromises strong consistency in favor of speed and throughput. In this paper, we investigate new forms of replication, termed *distorted replicas*, aiming at speeding up reads and writes in document-stores and aggregate-oriented NoSQL databases.

Document-stores avail the flexibility of structured documents (*e.g.* JSON) to pack closely related data in a single autonomous document, 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, prohibitive in a highly distributed environment. The concept of document is similar to the Domain-Driven-Design (DDD) pattern called *aggregate data model* [2]. Figure 1 depicts a slightly modified JSON-formatted document from the archives of the DBLP bibliography [3].

A key challenge in document-stores is how to model documents in order to meet an application needs in terms of performance and access patterns. Indeed, the whole document-store approach works well only when data access is aligned with the structure of documents [1]. If data is accessed in a different way, the whole system performance may be substantially impacted. Figures 2 and 3 illustrate two possible alternatives to the data model of figure 1. In the data model of figure 2, data is vertically partitioned in three sub-documents, tied up with the same *id*. In figure 3 data is organized by authors, rather than by publications. The data model of figure 2 is better suited for queries touching a small fraction of fields, as it avoids to load the whole fields in the memory hierarchy. Similarly, the document structure of figure 3 is better suited when data is accessed by authors.While with these simple examples it seems that the data models of figures 2 and 3 outperform the one of figure 1, it is just as easy to come up with examples where the data model of figure 1 is better.

In most cases, we cannot figure out in advance all use cases and data access patterns. Even if we do, it is hard, if not impossible, to find out a document structure capable of efficiently powering all possible queries. Relational databases have an advantage here as they allow to slice and dice data different ways for different queries [1].

Replication is ubiquitous in NoSQL systems. This paper proposes a new form of replication, termed *distorted replicas*, which leverages already existing replication to cope with the heterogeneity of data access patterns. The main idea behind distorted replicas is to restructure replicated data in different ways in order to provide the ability to explore it in different ways. While not physically identical, distorted replicas are logically identical and keep the fundamental property of being constructible from one another. Consequently, distorted replicas help in better supporting various data access patterns, while still ensuring high availability and fault-tolerance.

This paper proposes some basic distorted replication schemes allowing to boost I/O throughput, without additional costs. The proposed strategies are put into practice with MongoDB [4], but are general and applicable to almost all document-stores.

The remainder of this paper is organized as follows. Section 2 describes the main concepts related to document-stores. Section 3 introduces our distorted replicas policies. Section 4 presents related works. Section 4 gives an experimental study of distorted replicas performance. Section 5 concludes the paper.

2161-5330/17 $31.00 © 2017 IEEE
DOI 10.1109/AICCSA.2017.34

IEEE computer society

```
{"_id": "journals/vldb/KohlerLLZ16",
  "title": "Possible and certain keys for SQL",
  "author": [{"_id": "Henning Köhler"}, {"_id": "Uwe Leck"},
        {"_id": "Sebastian Link"}, {"_id": "Xiaofang Zhou"}],
  "citations": [ ], "pages": "571-596", "year": 2016,
  "url": "db/journals/vldb/vldb25.html#KohlerLLZ16",
  "ee": "http://dx.doi.org/10.1007/s00778-016-0430-9",
  "journal": "VLDB J.", "volume": 25, "number": 4}
```

Fig. 1. JSON-formatted document from the archives of the DBLP bibliography.

```
{"_id": "journals/vldb/KohlerLLZ16","pages": "571-596",
  "title": "Possible and certain keys for SQL",
  "url": "db/journals/vldb/vldb25.html#KohlerLLZ16",
  "ee": "http://dx.doi.org/10.1007/s00778-016-0430-9"
}

{"_id": "journals/vldb/KohlerLLZ16",
  "author": [{"_id": "Henning Köhler"}, {"_id": "Uwe Leck"},
   {"_id": "Sebastian Link"}, {"_id": "Xiaofang Zhou"}],
  "citations": [ ]
}

{"_id": "journals/vldb/KohlerLLZ16", "year": 2016,
  "journal": "VLDB J.", "volume": 25,"number": 4
}
```

Fig. 2. Vertically partitioned document.

II. 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 to discuss features that may be of interest in particular scenarios.

A. Data model design : normalization versus aggregation

Document-oriented databases store and retrieve data in the form of documents formatted in XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which consist of field-value pairs. A field value may be a scalar value, an array of scalar values, a document (*i.e.* embedded document) or an array of embedded documents. Documents with similar structures are typically grouped into *collections* [1]. Collections have no predefined schema and do not enforce document structure [5]. Nonetheless, collections should not be treated as heaps [6]. In practice, documents in a collection ought to share a common subset of fields, in order to comply to at least some common data access pattern and present a consistent and robust data interface to an application [7]. To help in solving this issue, some document-stores such as CouchDB and MongoDB, provide support for checking structural and domain constraints during updates and insertions.

One of the most important document-modeling choices is how to represent relationships between data : with references (normalized data model) or with embedded objects (denormalized data model). References represent the relationships between data by including a link from one document to another, just as in the relational model. With such a normalized model, 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 document and hence, avoids cross-nodes joins and writes. A document embedding documents nested inside it, is called in the sequel an *aggregate*. Aggregate is a term that comes from Domain-Driven Design [2] and refers to a cluster of associated objects which are treated as a unit for data manipulation and management. Dealing in aggregates makes it easier to operate on a cluster, since aggregates make natural units for sharding and replication [1].

```
{"_id": "Henning Köhler",
 "publications": [{"_id": "journals/vldb/KohlerLLZ16",
    "title": "Possible…", "pos": 1,… "author": […],…},
    {"_id": "journals/debu/KohlerLZ16","pos": 1,…},…]}
```

Fig. 3. Data organized according to a different aggregate root (*i.e.* by authors).

B. Replication

In clusters of hundreds, or even thousands, of geo-distributed commodity servers, high rates of hardware and network failures are common and the system should be able to instantly cope with them [7]. 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* in MongoDB. A replica set is composed by one master node, called *primary*, a set of slave nodes, called *secondaries* and one optional node called *arbiter*. 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 a special collection, called the $opLog$ (*i.e.* operations Log). Secondary nodes periodically import the opLog from the primary or from any other member of the replica set. Secondaries then apply all changes to their local replicated collections in such a way that they reflect the master collections [4]. In case of a master failure, an eligible slave holds an election to elect itself the new master [4]. When the node that failed comes back, it joins in as a slave and synchronizes its local collections. An arbiter server does not replicate the master collections. Its only purpose is to obtain a quorum during the election of a new master.

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. To ensure strong consistency, reads are by default sent to the primary and hence only access the most recent data copy. In such case, replication is only used for availability and fault-tolerance. If it is required to use replication for increased read throughput, a client application may choose to read from one of the existing slaves, with the risk of reading outdated data.

In contrast to MongoDB, some systems such as CouchDB and Amazon Dynamo allow a master-master replication, in which writes and reads are distributed over replicas. Distributing writes prevents from having a single server supporting all writes, but raises consistency issues, as separate servers may create conflicting versions [7].

C. Sharding

In most NoSQL systems, replication is not the primary mean for scaling and sharding is often a better strategy. 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 [4]. Typically, NoSQL systems do not support distributed joins and implement limited distributed transactions capabilities, due to the I/O overhead and coordination required [8].

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 server is added to the cluster.

The mapping between chunks and shards ($i.e.$ mapping metadata) need to be kept to be able to route subsequent read and write requests to the appropriate shard. Some shared-nothing architectured systems, such as Amazon Dynamo, duplicate some of the mapping metadata on each shard of the cluster. Other systems such as MongoDB store mapping metadata in a separate centralized server ($i.e.$ the *Config Server*).

In MongoDB each shard can be a complete replica set ($i.e.$ data is first distributed and then replicated). In addition to shards and config servers, a MongoDB sharded cluster is composed by a set of query routers, named mongos. When a mongos instance starts, it loads a copy of the config server database and route the reads and writes from client applications to shards.

III. Distorted replicas

Distorted replicas leverages already existing replication to efficiently solve the problem of access patterns heterogeneity. With distorted replicas, instead of having a document and its replicas physically identical, they are only logically identical. By logically identical we mean that data is organized differently in each replica, but the different replicas can be constructed from one another.

This section exposes the main consequences of the aggregation model and explores different basic schemes for solving them with distorted replicas. In the sequel we adopt the MongoDB architecture for its flexibility, but similar schemes can be implemented with other document-stores.

A. Vertical partitioning

1) Write/read overheads: 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. Consider for instance the relationship between a publication and its authors. Some applications will want to read the authors emails whenever they access a publication; this fits in well with combining the publication with its authors into a single aggregate. Other applications, however, want to access the email of an author and thus only access the author's data. 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. As aggregates are commonly large in size, loading large amount of data irrelevant for a given query may seriously wastes main memory and disk bandwidths and increases the amount of CPU cycles wasted in waiting for data loading [9]. The introduced write/read overhead is one of the most important downsides of the aggregate approach.

A more subtle limitation is that in some cases the update or the insertion of a field in a document, may lead to the entire document re-write. This happens when the update or the insertion of new fields causes a growth in the document size beyond the allocated space for that document. If space is pre-allocated for each document, the fragmentation of data files is avoided, but even with pre-allocation, updating documents gets slower as documents grow.

2) Principle: The aim of the first scheme of distorted replicas is to store a collection replica at a lower level of granularity to reduce the write/read overheads. It consists in partitioning vertically a base document with n fields in $m \leq n$ sub-documents, with the assumption that data of a particular sub-document will usually be accessed together. As illustrated in figure 2, each sub-document holds : (i) one or more fields; and (ii) the base document id identifying the original base document that the values came from. Vertical partitioning of a collection of documents, results in sub-collections which are similar to column families in wide-column stores ($e.g.$ Cassandra and Hbase). As in wide-column stores, subsets of fields are stored together and the different pieces of the same row ($i.e.$ document) are identified by the same id.

Vertical partitioning has poor space utilization as it stores the base document id along with each of the vertical partitions [10]. Additional space may also be lost as each partition is padded and stored with its own header. Vertical partitioning performs also poorly for insertions because multiple distinct (sub-) documents have to be inserted for each inserted base document.

Despite its drawbacks, vertical partitioning has one great benefit against the aggregate model : a query can only load the sub-documents that it requires, instead of loading the whole base document. This is especially helpful for workloads with high rate of updates or reads with low projectivity ($i.e.$ touching a small fraction of fields).

We should notice that the idea of maintaining two copies of the same data, one stored following the Decomposition

Storage Model [11] and the other following the N-ary Storage Model has been first introduced in [12] and applied to RDBMS running on servers with mirrored disks ($i.e$ RAID 1). [12] proposes many schemes to speed up joins of sub-relations. In our work we further extend the approach to document-stores running on a cluster of replicated servers. We are less interested by joining sub-documents, as in our case, joins should be performed only when we need to reconstruct the base replica from the vertically partitioned one. Operations accessing data from different sub-collections, have to be directed to the replica hosting the aggregated documents.

3) Sub-documents storage: Sub-documents resulting from vertical partitioning can either be stored in the same collection or in separate collections. If the sub-documents are put in the same collection, they cannot be identified with the id of the base document. The id must nonetheless be added to each of the sub-documents to allow the reconstruction of the base document. The id can also acts as the collection shard key to ensure that all the partitions originating from the same document fall within the same shard. The main strength of this scheme is that it allows to store the sub-documents sorted according to the document id. Having the sub-documents stored contiguously on the disk allows to efficiently access all the partitions of the same document and can be an alternative to joins if the base document needs to be reconstructed.

The main weakness of this scheme is that queries with low projectivity are constrained to access all the sub-documents and to perform a type test for each accessed one. Consider the example of figure 2 and a map/reduce job searching for the publications of each author. If the sub-documents are placed in the same collection, the map is forced to treat each sub-document, and, for each, to test whether or not it contains an "author" field.

Putting sub-documents in separate collections allows to only process sub-documents relevant for a given query and avoids type tests. The counterpart of this strategy is that it requires to loop over more than one cursor if more than one partition need to be scanned. Another important limitation is that in some systems, such as MongoDB, a map reduce job can only be processed over a single collection. As a consequence, performing a map reduce job over fields that belong to different sub-collections, may be tedious. In our opinion, putting the sub-documents in separate collections is better than storing them in the same one, as it avoids loading unnecessary data in the memory hierarchy. If a query or a map/reduce job needs to read data from more than one sub-document, it can be directed to the replica with the aggregate model.

B. Multiple aggregate roots

Objects in an aggregate are bound together by a root object, known as the *aggregate root* [2]. In most cases, there exist many root candidates. Using our DBLP example, publications, authors and conferences are all root candidates. Bounding objects by one of the candidate root may help with some data interactions, but is necessarily an obstacle for many others. As reported in [2], the whole aggregate orientation approach works well only when data access is aligned with document roots.

The idea behind the "Multiple Aggregate Roots" scheme is to use different aggregate roots for a collection and its replicas. Using our DBLP example, the base document collection may be rooted at publications, while one of its distorted replicas may be rooted at authors (see figure 3). This can be easily implemented by processing the database log (see section V).

If documents are rooted at publications, distorted replicas prevent from digging in every document to find out an author publications. Nonetheless, if the collection is sharded, distorted replicas do not prevent from visiting each shard. The number of accessed shards can be reduced by maintaining a Counting Bloom Filter (CBF) [13], [14] for each shard and host the filters within the metadata server ($e.g.$ the Config Server in MongoDB). As a regular Bloom Filter (BF), a CBF allows to quickly test whether an author appears in a given shard. Unlike a BF which associates a single bit to each of the filter buckets, a CBF associates a counter with a fixed size of c bits. The i^{th} counter indicates the number of elements currently hashed to the i^{th} bucket.

Our choice of using CBFs instead of BFs is motivated by data movements between shards, unavoidable to maintain a balanced data distribution. If documents from a replicated collection are migrated from one shard to another, the filters associated with the two shards, may need to be updated accordingly. Let consider the DBLP dataset (where data is organized by publications) and a distorted replica where data is organized by authors. Suppose that p is a publication that need to be migrated from a shard s_i to a shard s_j, and a one of its authors. Moving p results in : (i) the addition of a to the filter associated to s_j; and (ii) the deletion of a from the filter associated to s_i, if p were the only publication of a in s_i. While a BF does not allow deletions of elements, deleting an element from a CBF can be safely done by decrementing the relevant counters.

IV. Related work

The problem of heterogeneity in data access patterns has been addressed in different ways. This section focuses on approaches that use different physical designs for replicas of the same data collection.

A. Secondary indexes and the divergent physical design

In [15] the authors propose a novel tuning paradigm for replicated databases, coined *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 [15]. 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 [16] where the authors proposed RITA, an index-tuning advisor for replicated databases. RITA allows to : (1) generate fault-tolerant divergent designs; and (2) spread evenly the load over replicas.

As aptly stated in [1] with secondary indexes we're still "working against the aggregate structure". Indeed, by indexing documents on author identifiers, some of the latency would be hidden, but performance would only be sub-optimal, since the retrieved documents are typically scattered across the storage device. Secondary indexes are also not helpful for map/reduce jobs and for queries involving regular expressions. In some cases, scanning the whole collection is a better solution. Using our DBLP example, as authors may have variations in their first names ($e.g.$ "Mike Stonebraker", "Michael StoneBraker", etc.), it is common to use a regular expression to find out their publications. With the DBLP dataset and the MongoDB settings of section V, a query looking for the publications of any author having Stonebreaker as last name (authors._id: {Stonebraker$}) takes on average 306916 ms with a secondary index on authors ids, 34776 ms without an index and 8742 ms if data were organized by authors.

B. Divergent block layouts

[17] proposes Trojan Layout, a data layout inspired by PAX (Partition Attributes Across) [18] 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 miniblocks. The i^{th} miniblock stores all the values of the i^{th} attribute of R. The Trojan layout splits a HDFS block into $m \leq n$ miniblocks and stores in each miniblock the values of k ($1 \leq k \leq$) attributes ($i.e.$ vertical partitioning inside each chunk). Trojan Layout provides a high degree of spatial locality when the values of the k grouped attributes are sequentially accessed and avoids to read $(n-k)$ irrelevant attributes for a given query. To better handle a mix of queries with different access patterns, [17] proposes also to group attributes differently in each HDFS block (chunk) replica according to the query workload.

The Trojan Layout is helpful for queries that sequentially access some given attribute values (of a flat records), but it can't help with queries that need to access data according to a different aggregate root. As for secondary indexes, with the Trojan Layout we're still "working against the aggregate structure". Using our DBLP example, the Trojan Layout allows to group author values at the chunk level. However, it does not allow to group publications by authors at the shard level (which require a complete restructuration of the data). Thus, with the Trojan Layout, a query such as "find Stonebreaker's publications" will always require to perform a map/reduce Job and to process every single publication in the collection. With the multiple aggregate roots model the query only requires to visit one document in each shard where the author appears.

Note here that the multiple aggregate roots model and the Trojan Layout are not antagonist: one can use the first model to restructure data according to a different aggregate root and then store the restructured data with the Trojan layout to further improve queries that sequentially access some given attribute values.

C. Materialized views

In RDBMSs, a materialized view (MV) is a named query whose results are persisted. The primary purpose of materialized views is to optimize an expensive read query by precomputing and caching all or part of its intermediate results.

NoSQL systems, such as CouchDB, allow the creation of MVs (they are called views, but are more akin to MVs since their results are persisted). As in most NoSQL systems, a view in CouchDB is a persisted result of a map-reduce computation. This result takes the form of a new collection organized as a B-tree built on the id [7]. MongoDB provides a similar feature through the so called "incremental map-reduce". As CouchDB views, incremental map-reduce consists in persisting the result of a map-reduce job in a new collection that can be queried, replicated and sharded as a regular collection.

There are two rough strategies to refresh an MV: eager or lazy. With the eager approach, the MV is updated at the same time as the base data. This approach allows to keep MVs as fresh as possible. However, it may lead to an important write overhead, as each write operation needs to be instantly propagated not only to the document replicas, but also to all the MVs derived from the updated collection.

The lazy approach adopted by CouchDB, consists in refreshing an MV when it is queried and by re-evaluating the map-reduce only on changed documents [19]. Processing updates by batch at query time, reduces the write overhead, but may significantly slowdown reads. An alternate refresh scheme is adopted by MongoDB, where an incremental map-reduce is re-evaluated on demand. Such an approach may lead to situations where client applications read stale data.

As materialized views, distorted replicas allow to 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 MVs introduce a significant write overhead. Whether handled at the base data update time or shifted to the view query time, the write overhead may substantially impact performance.

It is worth noticing that if MVs 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 which uses MVs to locally replicate remote data in a replication environment. Oracle allows in addition to create "updatable materialized views" which enable to insert/update/delete data through an MV.

D. Pluggable storage engines

Some database systems, such as MySQL, Dynamo and MongoDB are shipped with different storage engines and applications are allowed to choose the one that is the most appropriate to their workloads. One of the most interesting features of MongoDB is that it allows the use of multiple storage engines within a single replica set ($i.e.$ the replicas of the same document can be stored and processed by different storage engines). The pluggable storage engines approach can be considered as such as a special case of distorted replication.

A shortcoming of the above approach is that document replicas remain physically identical ($i.e.$ have the same structure). Even with the most suitable storage engine, if a structure is not optimal for a given access pattern, performance can only be sub-optimal : the storage engine can at most hide some of the latency, but can not hide it all.

V. Performance Evaluation

Most of existing NoSQL benchmarks, such as [20], focus on NoSQL systems performance in terms of throughput and latency. They typically treat data as heaps of key-value pairs where values are opaque : queries are often limited to keys, read operations are assumed to retrieve the whole record fields and the update of a set of fields or of a single field are not distinguished.

In this work we are interested in bringing to light the potential benefits expected from restructuring the replicas of the same data in different ways. Consequently, we need to capture data semantics to identify root candidates and to perform vertical partitioning. To quantify the potential benefits of each distortion scheme, we used therefore a real dataset based on the DBLP Computer Science Bibliography [3] and a set of realistic workloads.

In the remainder, subsection V-A presents the tools used for synchronizing the proposed distorted replicas. Subsection V-B describes the main features of the DBLP data set. Subsection V-C defines the workloads considered and discusses the results.

A. Prototype for distorted replicas synchronisation

Whether distorted or not, replicas have to be kept synchronised when data is inserted, updated or deleted. In our work, we implemented a distorted replicas synchronizer based on the operations log and tailable cursors.

MongoDB keeps track of all operations that change the state of replicated documents in a special collection, called the opLog, stored in a special database called local (`Local.opLog.rs`). Each document in the opLog corresponds to a single operation performed on the primary node. OpLog documents contain several fields, including [21] : the timestamp when an operation was performed, the type of the operation performed, the name of the collection affected by the operation and the new state of the affected document.

As the write ahead log of most of RDBMSs, the opLog has a fixed size ($i.e.$ capped collection) and behaves like a circular queue : when it reaches its max size, the newest entries replace the oldest ones. The opLog collection can be queried just as a regular collection, from the Mongo shell or from any MongoDB driver. To synchronise our replicas without replaying already applied operations, it is necessary to only see the latest entries added to the opLog. MongoDB doesn't have triggers and automatically close a cursor when it exhausts its result set. To prevent from requerying over and over the opLog to get newly added entries, MongoDB uses and allows to define a special kind of cursors, called *tailable cursors*. A tailable cursor is a kind of listener that only shows new data as it is written to a collection. When a tailable cursor reaches the end of the result set, it is not closed. Rather, it sleeps and waits for more documents to be added to the collection. If additional documents are inserted, the tailable cursor retrieves them. Tailable cursors are the natural way to intercept opLog's new entries (MongoDB uses tailable cursors to tail the opLog [4]).

B. Dataset and settings

The DBLP Computer Science Bibliography, is a data set containing 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 with respect to its type and has an id field that uniquely identifies it. Ids resembles to slash separated Unix file names [3].

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 about 3.1 million publication documents embedding about 1.6 million authors. The average document size is 538 bytes and the whole collection size is 1.5 GB.

Two distorted replicas were dynamically generated using the synchronizer discussed in subsection V-A. The first one uses vertical partitioning as discussed in subsection III-A. This storage model is referred to as VPM in the sequel. With this scheme each base document is partitioned in three subdocuments as illustrated in figure 2. The average object size in each of the three sub-collections is respectively 254 kB, 191 kB and 102 kB.

The second distorted replica organizes data by authors as discussed in subsection III-B. We added a pos field to each author in order to keep the order in which an author appears in a given publication. The resulting distorted replica is composed by 1.6 million documents having an average size of 2.3 MB. In the sequel, the aggregate data model where data is organized by pubications is referred to as ADM1 and the one where data is organized by authors as ADM2.

Simulations were performed on a dedicated dual core i5-3230M system, 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. This computer features 4 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. Experiments were conducted on MongoDB release 3.2 and its storage engine WiredTiger. MongoDB was run using its default settings and no special tuning was done. As the Mongo shell does not allow to manipulate simultaneously documents from different collections, we conducted our tests through the Java driver. All reported times are the average of three consecutive runs.

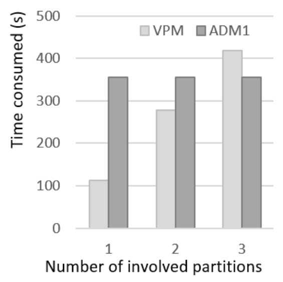

Fig. 4. Scan cost as function of projectivity.

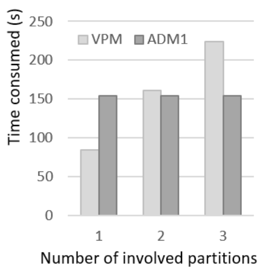

Fig. 5. Query cost as function of projectivity.

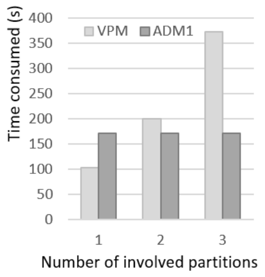

Fig. 6. Update cost as function of projectivity.

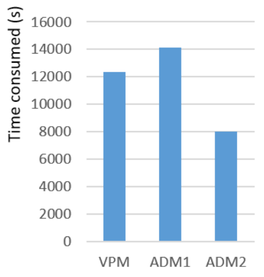

Fig. 7. Average time consumed when searching for the co-authors graph.

C. *Workload and results*

1) *Relevancy of vertical partitioning:*

a) Insertion Cost: The insertion cost is measured by the average time elapsed during the insertion of 3.1 millions documents. It equals 1714 s with ADM1 and 3854 s with VPM. As expected, insertions are faster with ADM1. This is due to the fact that a single write suffices to push all the fields of a document, while with VPM, the server handles write requests for multiple collections which are likely scattered across the disk. The performance of VPM should be worst if the base collection were decomposed in more than 3 partitions.

b) Scan cost: Figure 4 depicts the average time consumed by a full collection scan as function of projectivity (*i.e.* number of involved partitions). The only case where ADM1 outperforms VPM is when all sub-collections need to be scanned.

c) Query cost: To evaluate the query cost, we considered three query types. The first type represents queries that only touch fields belonging to the same sub-collection. The second type (resp. the third type) represents queries that requests fields belonging to 2 sub-collections (resp. 3 sub-collections). Type 2 queries are equivalent to joins over 2 sub-collections. Type 3 queries are equivalent to the reconstruction of base documents from their sub-partitions. For each query type, 100K publication `_ids` were randomly selected using the `$sample` operator of MongoDB. For each `_id`, we measure the average time needed to extract the corresponding fields, from one sub-collection, two sub-collections and three sub-collections. The obtained results are compared to the time consumed to extract the same fields from the base collection.

As shown in figure 5, ADM1 performance are stable. This is due to the fact that the whole document need to be extracted, regardless the number of requested fields. For type 1 queries, VPM consumes on average 48% less time than the aggregate model. For type 2 queries, ADM1 performance are equivalent to VPM. For type 3 queries ADM1 is 38% faster than VPM.

d) Update cost: As for the query cost, to evaluate the update cost we consider 100K randomly selected publication `_ids` and three update types. Updates of type 1 (resp. type 2 and 3) involve scalar fields belonging to one partition (resp. 2 and 3 partitions). The considered updates do not grow the document size. When the updated fields belong to one partition, VPM is 40% faster than ADM1. As shown in figure 6, VPM performance deteriorates if fields from more than one partition are updated at a time. In our experiments, ADM1 is 46% faster than VPM when 3 partitions are involved. We should notice here that in practice updating all, or almost all of the fields of a record at a time is unusual. In such case, a delete followed by an insert provides generally better results.

e) Map/Reduce: To illustrate the behavior of VPM and ADM1 with regard to map reduce jobs, we consider the classical case consisting in finding the co-authors graph : for each pair of authors who have at least collaborate on a publication, we calculate the total number of publications they participated in. As expected VPM outperforms ADM1, as it only loads the author field, while ADM1 loads all the fields into the memory hierarchy.

2) *Relevancy of the multiple aggregate roots scheme:*
To give an order of magnitude of the gains that could be drawn from structuring data with different aggregate roots, we considered 5 realistic queries/map-reduce jobs, generally simple to process with one replica, but tedious with the other.

a) Query 1: "Find the publications of each author": If data is organized by authors, a simple `find()` from the mongo shell or the Java driver would be sufficient to get the publications of each author (less than 1 ms). However, when data is organized by publications, it is necessary to perform the map/reduce job, consuming on average 8155 s (from the Mongo shell).

b) Query 2: "Find the authors of each publication": Inversely to Query 1, a simple find operation is sufficient (less than 1 ms) if data is organized by publications and a map/reduce job is necessary if data is organized by authors. The average time required to process the map/reduce job is 7873 s (from the Mongo shell).

c) Query 3: "Find the publications of a given author": In our parsed DBLP dataset, the average number of publications per author is 5.46. We randomly selected 3 authors having each 6 publications. When data is organized by authors, such a query consumes less than 1 ms. When it is organized by publications it consumes on average 29 s (from the Mongo shell).

d) Query 4: "Find the authors of a given publication": The average number of authors per publication in our parsed DBLP dataset is 2.85. We randomly selected 3 publications co-written by 3 authors. If data is organized by publications, finding the authors of a given publication is executed in less than 1 ms. When it is organized by authors, it requires on average 25 s (from the Mongo shell).

e) Query 5: "Find the co-authors graph": As illustrated in figure 7 when data is organized by authors, the map/reduce job consumes on average 35% less execution time than when it is organized by publications.

f) Sharded setup: In this set of experiments, we considered a very simple sharded setup, where the documents of the collection publications are distributed over two mongod instances, running on the same machine. Each of the two shards holds a sub-set of the base documents and the corresponding distorted replica, containing the same data organized by authors. The first shard contains 1.769.638 publications and their corresponding 1.059.357 authors. The second shard contains 1.346.159 publications and 988.934 authors. We considered a simple query searching for the publications of each author in the data set. A Counting Bloom Filter (CBF) is associated to each shard in order to indicate which authors appear in which shards. Without CBF, a query is systematically sent to both shards, and the returned results are merged. With CBF, the query is only sent to the shard where a given author appears. The average execution time is 2387 s with CBF and 2983 s without (from the Java driver). Even with our simplistic shard setup, CBF help in reducing the query execution time by mean of 20%. We should notice here that the gain allowed by CBF would be higher if the shards run on different machines, as unnecessary requests need to be transferred across the network.

VI. Conclusion

Aggregates are useful in that they pack into one document, data that is expected to be accessed together. Aggregates are essential to running on a cluster, but severely limit the ways data can be efficiently explored and processed.

This paper introduced distorted replicas, a new replication scheme helping in better handling data access heterogeneity in document-stores. Distorted replicas take advantage of already existing replication to restructure the same data in different ways. By doing so, distorted replicas provide new ways for exploring data, while still preserving the ability to reconstruct replicas from one another.

The paper studied basic distortion schemes. In the first one, replicated data is vertically partitioned. In the second one, replicated data is organized according to a different aggregate root. Based on the workload mix, the complexity of queries, and the update frequency, an application can direct its reads and writes to the distorted replica that is most suitable. We showed that even with simplistic schemes, performance can be substantially improved.

The main goal of this work was to bring to light the relevancy of distorted replicas and to motivate research in this direction. We implemented for this purpose a prototype for synchronizing distorted replicas, based on the operations log available in most NosSQL systems. This prototype is sufficient as a proof of concept, but many challenging problems need to be thoroughly thought out. As a part of our future work, we intend to further examine the dynamic migration of data between shards and its impact on distorted replicas and on load balancing. This especially holds when data and its replica are arranged according to different aggregate roots. In this paper we recommend the use of counting bloom filters to ease data movements between shards, but more sophisticated schemes can be implemented. Another interesting point to consider is the definition of a cost model for each type of data organization, to help query optimizers in determining the replica to which a query should be directed.

References

[1] P. J. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 1st ed. Addison-Wesley Professional, 2012.

[2] E. Eric, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] M. Ley, "DBLP – Some Lessons Learned," *PVLDB*, vol. 2, no. 2, pp. 1493–1500, 2009.

[4] "Mongodb," http://www.mongodb.com, [Accessed: 2016-09-01].

[5] R. Arora and R. R. Aggarwal, "Modeling and Querying Data in MongoDB," *International Journal of Scientific and Engineering Research*, vol. 4, no. 7, pp. 141–144, 2013.

[6] L. Wang, S. Zhang, J. Shi, L. Jiao, O. Hassanzadeh, J. Zou, and C. Wangz, "Schema Management for Document Stores," *Proc. VLDB Endow.*, vol. 8, no. 9, pp. 922–933, May 2015.

[7] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart, *Web Data Management*. Cambridge University Press, 2011.

[8] "Oracle nosql database concepts manual," http://docs.oracle.com/cd/NOSQL/html/ConceptsManual/index.html, [Accessed: 2016-09-01].

[9] K. Jouini, G. Jomier, and P. Kabore, "Read-optimized, cache-conscious, page layouts for temporal relational data," in *Proc. of the 19th Int. Conf. on Database and Expert Systems Applications*, ser. DEXA '08, 2008, pp. 581–595.

[10] K. Jouini and G. Jomier, "Modèles de stockage orientés interrogation pour bases de données temporelles," *Ingénierie des Systèmes d'Information*, vol. 15, no. 1, pp. 61–85, 2010.

[11] G. P. Copeland and S. Khoshafian, "A Decomposition Storage Model," in *ACM SIGMOD'85*. ACM Press, 1985, pp. 268–279.

[12] R. Ramamurthy, D. J. DeWitt, and Q. Su, "A Case for Fractured Mirrors," *The VLDB Journal*, vol. 12, no. 2, pp. 89–101, 2003.

[13] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[14] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," in *ACM Int. Conf. on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. ACM, 2014, pp. 75–88.

[15] M. P. Consens, K. Ioannidou, J. LeFevre, and N. Polyzotis, "Divergent physical design tuning for replicated databases," in *ACM SIGMOD Int. Conf. on Management of Data*, ser. SIGMOD '12, 2012, pp. 49–60.

[16] Q. T. Tran, I. Jimenez, R. Wang, N. Polyzotis, and A. Ailamaki, "Rita: An index-tuning advisor for replicated databases," in *Proc. of the Int. Conf. on Scientific and Statistical DB Mgmt*, ser. SSDBM '15, 2015, pp. 22:1–22:12.

[17] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: Right shoes for a running elephant," in *Proc. of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11, 2011, pp. 21:1–21:14.

[18] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *VLDB'01*. Morgan Kaufmann Publishers Inc., 2001, pp. 169–180.

[19] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous View Maintenance for VLSD Databases," in *ACM SIGMOD'09*. ACM, 2009, pp. 179–192.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *ACM Symposium on Cloud Computing*, ser. SoCC '10. ACM, 2010, pp. 143–154.

[21] K. Chodorow and M. Dirolf, *MongoDB: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2010.