Noname manuscript No.
(will be inserted by the editor)

A Cost Model for Random Access Queries in Document Stores

Moditha Hewasinghage[1] · Alberto Abelló[1] · Jovan Varga[1] · Esteban Zimányi[2]

Received: date / Accepted: date

Abstract Document stores have become one of the key NoSQL storage solutions. They have been widely adopted in different domains due to their ability to store semi-structured data and expressive query capabilities. However, implementations differ in terms of concrete data storage and retrieval. Unfortunately, a standard framework for data and query optimization for document stores is nonexistent, and only implementation-specific design and query guidelines are used. Hence, the goal of this work is to aid automating the data design for document stores based on query costs instead of generic design rules. For this, we define a generic storage and query cost model based on disk access and memory allocation that allows estimating the impact of design decisions.

Since all document stores carry out data operations in memory, we first estimate the memory usage by considering characteristics of the stored documents, their access patterns, and memory management algorithms. Then, using this estimation and metadata storage size, we introduce a cost model for random access queries. We validate our work on two well-known document store implementations: MongoDB and Couchbase. The results show that the memory usage estimates have the average precision of 91% and predicted costs are highly correlated to the actual execution times. During this work, we have managed to suggest several improvements to document storage systems. Thus, this cost model also contributes to identifying discordance between document store implementations and their theoretical expectations.

Keywords Document stores · Cost model · Query · NoSQL

Moditha Hewasinghage[1]
E-mail: moditha@essi.upc.edu

Alberto Abelló[1]
E-mail: aabello@essi.upc.edu

Jovan Varga[1]
E-mail: jvarga@essi.upc.edu

Esteban Zimányi[2]
E-mail: ezimanyi@ulb.ac.be

[1]Universitat Politècnica de Catalunya, BarcelonaTech Barcelona, Spain
[2]Université Libre de Bruxelles 1050 Bruxelles, Belgium

1 Introduction

In the last couple of decades, NoSQL systems were introduced as an alternative storage mechanism to traditional Relational Database Management Systems (RDBMS). As of today, there are more than 100 NoSQL implementations, categorized into four main types, namely, key-value stores, document stores, column stores, and graph stores [3]. Among these, document stores have been one of the most popular, mainly because of the schema flexibility they provide.

Traditional RDBMS arrange data in tables with a fixed schema, and each tuple within the table adheres to it. Similarly, document stores arrange data in collections, and as the name suggests, they use document formats such as XML or JSON as the unit of storage. However, in contrast to RDBMS tables, the schema of a collection is not fixed and allows semi-structured data. This means the documents within a collection need not be homogeneous, and in extreme cases, one might have a collection containing documents that are entirely different from one another.

Motivated by various commercial needs, the industry drives most of the document store development. The

This is a post-peer-review, pre-copyedit version of an article published in *The VLDB Journal*.
The final authenticated version is available online at: http://dx.doi.org/10.1007/s00778-021-00660-x

semi-structured storage nature of document stores enables end-users to follow a data-first approach rather than a schema-first one. This can reduce the time and effort of defining the schema, especially for cases where there is some uncertainty about the data structure and content. However, in contrast to RDBMS, there is no formal standard, such as SQL and normalization theory for document stores. Instead of a formal data and query design, existing document stores provide guidelines for implementation-specific optimizations [13]. Furthermore, all document stores use primitive approaches to determine the optimal query execution plan. For example, MongoDB tries to execute all viable query plans that use indexes in parallel to finally choose the winning option. Here, rather than performing all the query plans, some plans that take more work units (execution time or the number of documents examined) than the currently leading one are discarded prematurely as an optimization. The winning plan is cached and used on similar subsequent queries. [1] Therefore, it is crucial to have a better schema design to support this query execution model.

There are no existing methodologies to evaluate the viable document store schema designs (referred to as design from now on), but only through expensive trial and error one could determine which is the best design. Here, a formal cost model would allow end-users to estimate how different design decisions affect performance rather than relying on their intuition. Relational cost models [14, 18, 25] are per-query and based on the current state of the DBMS at the time of query execution. The primitive approaches of document store query planners do not even rely on such information. Existing cost models in RDBMS are based on disk access as the number of disk accesses will always dominate the execution time of a query. Moreover, as a result of hardware cost reduction in recent times, most of the data is pushed into the memory for faster performance, and the cost models have adjusted accordingly [20]. Thus, most document stores encourage having the working dataset in memory to achieve higher throughput and lower response time. Deciding on which fraction of the data should be in memory depends on access patterns and memory allocation policies. Nevertheless, both of these systems need to persist data into the disk (there are in-memory systems that are out of the scope of our paper), and in most of the cases during the actual operations, most of the data does not fit into the available memory. Therefore, it is highly probable that the data is accessed from both memory and disk simultaneously. Hence, in our work, *we aim to predict the behaviour of the document store memory and its effect on query execution depending on data design decisions.*

In this paper, we present a generic cost model for random access in document stores based on storage metadata and memory usage. Both these parameters are specific for different document store implementation decisions. Metadata can be easily obtained from the data stores and depends on the disk storage structures. However, memory usage depends on memory mapping, associativity, and eviction policies, and each of them consists of several possibilities (e.g., pre-determined or shared associativity). Therefore, we introduce formulas for different possibilities and depending on the underlying document store, we pick the corresponding formulas to estimate the memory usage. Finally, we use these memory estimates in our generic cost model. We use Couchbase Server and MongoDB as two exemplars with different memory usage patterns to validate our cost model. Couchbase Server has pre-determined memory associativity per collection and maps individual documents into memory, whereas MongoDB has a single memory shared among the collections governed by a Least Recently Used (LRU) cache eviction policy and maps disk blocks to memory.

The objective of our work is not to predict an exact runtime, but rather to obtain a relative cost for a specific query under a fixed memory, workload, and varying design decisions (e.g., average document size, number of documents, and access frequency). Thus, the main contribution of this work is *a generic cost model for document stores for random access queries, which includes a detailed memory distribution estimation model for different memory management choices.* Based on experiments, we show that we are able to estimate memory distribution within an average precision of 91% and successfully predict the relative cost of queries concerning the most relevant parameters. To the best of our knowledge, this is the first attempt at a cost model, allowing us to predict the design impact in document stores. Even though it is possible to introduce this cost model into the native query processing of the document store implementation, it may introduce significant overhead hindering the performance. The simplicity of the current primitive query processing approaches is one of the many reasons behind the performance gains in document store or NoSQL systems in general compared to traditional RDBMS. Thus, we instead intend to use this cost model as a tool to enhance the schema design process, which is mostly rule-based. Using our approach, we can calculate a relative cost for a particular design under a predefined workload. This value can then be used as a measure of optimality together with other contradicting requirements such as storage space of a

[1] https://docs.mongodb.com/manual/core/query-plans

collection on each of the viable designs to select the best one. Thus, this is an initial step towards optimizing resource usage using a systematic data design. We have already implemented DocDesign [12], a decision aid system to determine optimal document store schema using the cost model discussed here. DocDesign is capable of evaluating alternative schemas for a particular use case and workload. Determining the relative query performance is a key component of DocDesign, and this cost model is essential for its functionality.

Document stores attract users who engage in rapidly changing requirements taking away the burden of fixing a database schema. Following this trend, the SQL:2016 standard has incorporated JSON into the specification allowing RDBMS to have unstructured data in a single attribute [22]. Therefore, data design for RDBMS also goes beyond strict normalization and might require a cost-based schema design and could benefit from our cost model as well. We were also able to identify several inconsistencies of existing document store implementations and propose how to improve them. Therefore, this work can also be used to *identify discordance of document store implementation behavior against a theoretical expectation*.

This paper is organized as follows. First, in Sect. 2, we explain the necessary background on cost models and discuss related work. We introduce and formalize the cost model together with generic and specific components in Sect. 3. In Sect. 4, we apply the cost model using MongoDB and Couchbase Server as examples. Then, we validate our cost model and memory estimation with extensive experiments in Sect. 5. Finally, we conclude our work in Sect. 6.

2 Background and Related Work

Most database system has its own cost model to determine query costs and make execution plans accordingly. RDBMS have been around for more than three decades and have well-established cost models and query planning capabilities [14, 18, 25]. These cost models mainly depend on the disk I/O as it is the most costly resource compared to other factors such as CPU calculations and memory access. The main considerations of these cost models are physical data structures, access paths, and the algorithms used.

Several works have been carried out on optimizations and cost models for XML databases [9, 16] as well as for object-oriented databases [2, 8]. Moreover, Manegold et al. proposed a generic technique to create cost functions for database operations in hierarchical memory systems in [20]. This approach claims to be extensible to include disk I/O. As this hierarchical memory model approach is most comparable to ours, we use it as a baseline for comparison with details in Sect. 5.4.

Contrary to classical RDBMS systems, in-memory databases do not depend on accessing data from the disk. Consequently, the cost models depend exclusively on CPU cycles, memory capacity, line size, and associativity [7]. As mentioned before, document stores utilize both disk and memory for optimized performance. Between the two, the disk access cost is several magnitudes higher than of the memory. Thus, the determining factor for the performance of a query will always be disk I/O when the data does not fit in memory (most of the cases). Hence, we base our cost model on the RDBMS approach due to the similarity in storage structures used in the observed document stores and the maturity of the approach. Furthermore, our cost model incorporates memory distribution in the calculation as needed for overall design optimization.

Data in a typical RDBMS is stored with a primary index that utilizes B-tree as the data structure when storing data in the disk. The internal nodes of the B-tree contain the indexed values, and the leaf nodes contain either data or pointers to data depending on the type of index (clustered and non-clustered). These nodes are stored as fixed-size blocks in the disk. The total size of a table depends on the record size, the number of records, and the indexing mechanism used. The data access paths can be identified as a table scan, random access, searching for one or several tuples, insertion, and deletion of a tuple.

A given query can have multiple execution plans subject to the access path to the tuples, index structures, and the order and execution algorithm for join operations. RDBMS use a cost-based estimation where alternative query plans are generated, and intermediate result sizes and cardinalities are estimated from the available statistics. Next, a cost is estimated for each plan based on blocks read and written to choose the best one. For example, in PostgreSQL, each of the operations has predefined cost value[2] and the alternatives are evaluated through genetic optimization.[3]

Caching is an essential concept in modern disk-based computer systems that refers to keeping already used data in memory so that the information can be served faster for future requests. The retained data originates from a prior request or calculation. When the data is fetched from the cache, it is considered a cache hit or, in the opposite case, a cache miss. In the case of a cache miss, the data needs to be fetched from the disk, which increases the latency. Thus, higher cache hit rates

[2] https://www.postgresql.org/docs/12/static/runtime-config-query.html

[3] https://www.postgresql.org/docs/12/geqo-pg-intro.html

lead to better performance [27]. However, the cache is generally smaller than the information that needs to be retrieved by an application. Therefore, cached data needs to be replaced over time. There are several eviction policies such as first-in-first-out, last-in-first-out, least recently used, etc [21]. Different cache policies have their strengths and weaknesses, and they are used depending on the application requirements [28]. In any case, it is essential to know the hit rate to analyze the performance of end-user applications. However, this is not trivial due to the complex nature of the cache, the replacement policies, and the access patterns. Frank King III introduced a Markov-chain-based cache hit rate approximation using the independent reference model [17]. Based on this approach, several other works have been conducted on approximating the cache miss and hit ratios for different cache policies [10,15,31]. These approaches provide an approximation of hit rate and cache usage not only for a single application but also for shared cache among several applications [6]. In our work, we followed a probability-based approach to have a simple model with lower runtime complexity.

The structure of the data also plays a vital role when implementing a cost model. Although document stores and RDBMS maintain structured data, they differ in several ways. Data in RDBMS is stored in and conforms to a user-defined structure (table), and a single real-world object may often span several tables. Document stores keep documents in a collection, but the structure of the documents within it can differ from one to another. Moreover, document stores support nested data structures and encourage denormalization, so that all of the information for a real-world object can be a single document in the database. Similar to key-value storage, data in document stores are kept with a primary identifier. However, unlike in key-value storage, the internal structure of data is not entirely hidden to the document store. Consequently, document stores provide more extensive query capabilities for the end-user.

RDBMS satisfy ACID properties and use approaches such as write-ahead logging to guarantee validity even in the event of errors. However, despite the fact that the data is still stored in the disk, document stores encourage keeping the working set in memory, so all the updates are done in transient storage and only periodically synchronized to the disk (most of the NoSQL systems use a similar approach to improve performance at the expense of reliability). Document stores have begun to include mechanisms such as logging protocols and transaction management to ensure reliability. How data is stored, what additional metadata is used, and the memory usage differs from one document store to another. Thus, we focus our study on two representative document store platforms: Couchbase Server and MongoDB.

Several works have been carried out on cost-based schema design for NoSQL systems. In NoSE, the authors evaluate alternative designs for column stores (Cassandra) using the cost of accessing different column families to answer the queries [23]. However, when it comes to document stores, estimating the cost becomes more complex due to the introduction of secondary indexes. The work by Vajk et al. [30] generates multiple alternative schema sketches using denormalization, starting with 3NF. These schemas are then evaluated based on the storage and the number of transactions to choose optimal cloud-based storage solutions. Mortadelo [5] uses a meta-modeling approach to generate database implementations from a high-level conceptual model for document and column stores. A query merging approach is used to optimize the access patterns to determine the optimal designs.

Regarding document stores, some work has been carried out on optimizing the data storage in Solid State Drives [26], storage-specific data models [32], and use case comparisons [11], but to the best of our knowledge, not much work has been done regarding the cost models especially for JSON-based storage systems. For example, MongoDB has a query planner for optimizing complex queries that consider multiple execution paths as explained before, and caches the winning plan for subsequent queries. Nevertheless, a server restart or adding/deleting an index will clear the cached query plans. Therefore, the present paper proposes a cost model that predicts not only relative execution time but also memory usage patterns independent from query plan caching that can be used to guide document design. In this context, our DocDesign [12] is a design support system that uses the cost model discussed in this work. The end-user can evaluate several candidate schemas for a given use case and workload in terms of storage size and query performance to determine the optimal one. The cost model is the most crucial component in determining the query performance under a given memory limit and query frequencies.[4]

3 Formalization of the Cost Model

In this section, we present our cost model based on the data storage and query mechanism of document stores. The cost model consists of a generic and a specific component. The generic component is based on three parameters, as shown in Fig. 1. First, the type of

[4] A demonstration of DocDesign is available in https://www.essi.upc.edu/dtim/tools/DocDesign

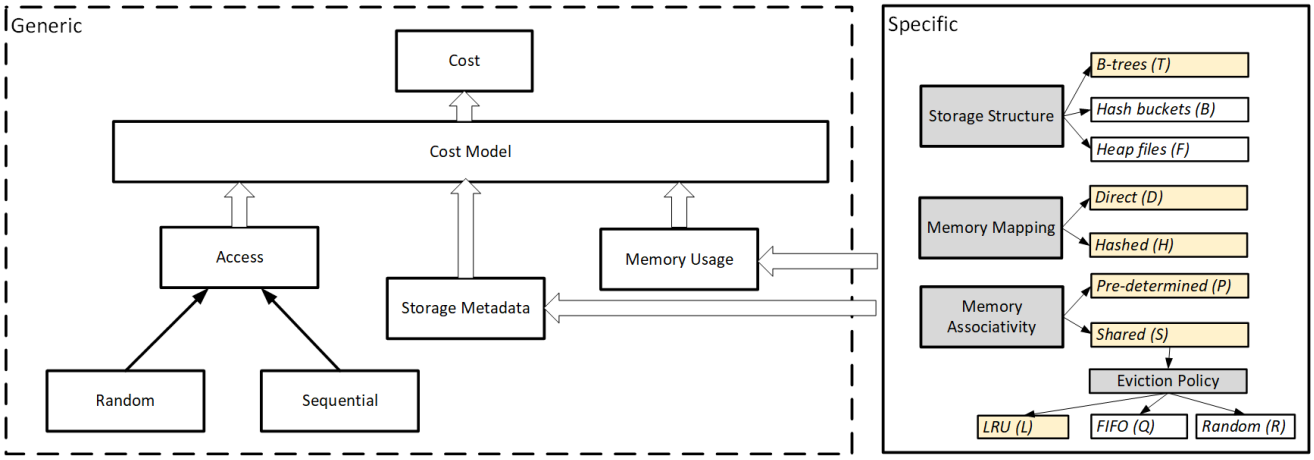


Fig. 1: Overview of the cost model for document stores

access affects cost. It can be random access through a primary or secondary index, or a sequential one where the entire collection is read. Second, the storage metadata characterizes the available data in terms of storage structure, indexes, and their sizes. Finally, the memory usage affects query performance, too. Among these parameters, the storage metadata and memory usage are directly affected by the underlying document store specifics. Therefore, we introduce the document store specific component consisting of three segments (storage structure, memory mapping, and memory associativity) to calculate and provide the two parameters to the generic component.

In our work, we focus on random access, which refers to accessing a document in a collection via physical disk location as these are the majority of the queries used on document stores (owing to the support of secondary indexes). The capital letters in italic correspond to the concepts in Fig 1. The storage size of a collection and its indexes mainly depend on the *physical storage structures*, which is the first segment of the specific component. These could be B-trees (T), hash buckets (B), or heap files (F) [18, 25]. In our validation, we use MongoDB and Couchbase Server in which B-trees are used. When bringing the data from the disk, it is directly mapped (D) into the memory following the structure as in the disk (MongoDB). However, Couchbase Server uses a hash-based mapping (H) where each document in the disk is brought and identified in the memory through a hash value indicating an in-memory bucket. Hence, we introduce *memory mapping* as the second segment of the implementation-specific component. The *memory associativity* in the presence of different collections can be: Pre-determined if the amount of memory used by each of the collections is also predetermined (fixed by the user in Couchbase); or shared if the usage of each collection determines how much memory is devoted to it (MongoDB). The proportion of documents in memory can be easily calculated in the case of pre-determined associativity (P). However, with shared associativity (S), this proportion is affected by the probability of accessing a collection and its storage metadata as well as the *eviction policy* being used. The eviction policy can be LRU (L), FIFO (Q), Random (R), etc.

We introduce an encoding for the external parameter configurations for the sake of convenience. The encoding contains three to four letters, each representing a segment in the order of storage structure, memory mapping, memory associativity, and eviction policy (only under shared memory). In this work, we introduce cost formulas for a B-tree based disk storage (T) under direct memory mapping (TD) with pre-defined associativity (TDP) or shared associativity under an LRU eviction policy ($TDSL$), and hashed memory mapping (TH) with pre-defined associativity (THP) or shared associativity under an LRU eviction policy ($THSL$). We validate our model using Couchbase Server (THP) and MongoDB ($TDSL$), introducing specific details regarding their implementation. As such, we can include other document stores in our cost model, depending on their configuration.

3.1 Generic Component

We assume that the data is retrieved from the disk to the cache and served for processing from the cache. Furthermore, the disk and the cache are accessed in fixed-size blocks, and the costs of reading a block from the cache or the disk are different but constant. The main focus of the present work is to estimate the cost for data

Table 1: Variables of the Cost Model

M	Total memory available for the document store	$B_{i_f}(C)$	Total index size on field f in blocks
f	Indexed field of a collection	$Req_f(C)$	Total number of requests to an indexed field
$Bsize_d$	Block size for data	$P_d(C)$	Probability of queried data block being in the cache
$Bsize_i$	Block size for index	$P_{i_f}(C)$	Probability of queried index block on field f being in the cache
$M_d(C)$	Memory blocks used for the data of a collection	$Cost_{Rand}$	Relative cost for a random read
$M_{i_f}(C)$	Memory blocks used for the index of a collection	$Rep_f(C)$	Number of repetitions of an indexed field
$T_{m/d}$	Time to read a block from cache/disk	$M_{alloc}(C)$	Memory allocated to a collection
$\mathbb{Q}$	Workload	$P(C,q)$	Probability of querying a collection by a query q
N	Number of collections	$\|Q\|$	Overall number of queries in an eviction cycle
C	A collection	$SF_f(C)$	Probability of a document being requested using field f
$Size_d(C)$	Average document size of a collection	$P_d^{req}(C)$	Probability of data block in cache being requested
$Size_{i_f}(C)$	Average index entry size of a collection (on field f)	$P_{i_f}^{req}(C)$	Probability of index block in cache being requested
$\|C\|$	Number of documents of a collection	$M_d^{sat}(C)$	Memory blocks used for the documents of a collection at saturation point
F	Fill factor of the B-tree	$M_{i_f}^{sat}(C)$	Memory blocks used for the index of a collection at saturation point
$R_d(C)$	Average number of documents in a block	K	Total size of non-leaf nodes of all the B-trees
$E_f(C)$	Number of unique requests to an indexed field	P_d^{block}	The probability that any of the documents in a leaf block being requested
$R_{int}(C)$	The average number of reference entries in an internal B-tree block	P_d^{block}	The probability that any of the documents in a leaf block being requested
$R_{i_f}(C)$	Average number of entries in a block for an index over field f	$Shots_d^{in}(C)$	Number of queried data blocks that are in memory within a time window (hits)
$B_d(C)$	Total collection size in blocks	$Shots_d^{out}(C)$	Number of queried data blocks that are not in memory within a time window (misses)
$Bsize_{int}(C)$	Internal B-tree block size	$Size_{int}(C)$	Internal B-tree reference entry size
$Mult_{i_f}(C)$	Multiplying factor of a secondary index	$M_{user}(C)$	Memory allocated by the user for a collection C in pre-defined memory associativity

access via the primary or secondary index of a collection (random access). We exemplify our approach on B-tree storage used by both Couchbase and MongoDB. For the sake of simplicity, we assume that the internal nodes of the B-trees are never removed from the cache in the calculations. Nevertheless, in case that the size of the internal nodes is relevant compared to the amount of memory available, extending the formulas to include them is straightforward (detailed explanation in Sect. 4).

Data distribution plays a vital role in document stores. Our estimations are intended for a single instance of a document store. However, they can be extended into a distributed environment. Assuming that the data has a uniform distribution among the nodes, we can estimate the size of the collections and indexes lying on each node by merely dividing the number of documents by the number of nodes. Therefore, the formulas can be directly applied to each of the nodes independently, just considering that distribution introduces an additional network cost. Existing work has added this cost together with the I/O cost as a weighted sum [24]. The network cost can be estimated by the size of data that needs to be moved between the nodes depending on the query. For example, if a query contains a shard key in MongoDB it will execute it only on the relevant nodes, but if it does not, the query needs to be performed on all the nodes, and the results need to be

aggregated.[5] Random access queries can hardly benefit from data distribution, as the end result is accessing a single piece of data in a given machine through a given path. We compared the runtime of a sharded cluster against a single instance for the experiments carried out in this work. In all the cases, the runtime of the distributed system was more than that of the single instance due to the added network cost, except for those instances that can accommodate all the data in memory with more machines.

Table 1 lists the variables used in the cost model equations. We define the number of cached data blocks as $M_d(C)$ and cached index blocks as $M_i(C)$. These numbers and their behavior vary depending on the type of document store and the access pattern of a collection, but we assume that the index entry and the document sizes are smaller than the block size ($Size_{i_f}(C), Size_d(C) \ll Bsize_d, Bsize_i$) and the blocks are filled in the average up to a percentage F. Thus, the average number of documents in a document block of collection C, $R_d(C)$, and the average number of index entries in an index block (on a particular field f) $R_{i_f}(C)$ can be defined as follows.

$$R_d(C) = F \cdot \left\lfloor \frac{Bsize_d}{Size_d(C)} \right\rfloor \quad R_{i_f}(C) = F \cdot \left\lfloor \frac{Bsize_i}{Size_{i_f}(C)} \right\rfloor \tag{1}$$

Now, we can define the total data leaf blocks of a collection $B_d(C)$ and the total index leaf blocks of the collection $B_{i_f}(C)$ dividing the number of documents by the respective number of documents and index entries that fit in a block as follows. For a secondary index, the leaf level entries depend on how many documents are being pointed by a single index entry, specified by a multiplying factor $Mult_{i_f}(C)$.

$$B_d(C) = \left\lceil \frac{|C|}{R_d(C)} \right\rceil \quad B_{i_f}(C) = \left\lceil \frac{|C| * Mult_{i_f}(C)}{R_{i_f}(C)} \right\rceil \tag{2}$$

If there are $M_d(C)$ data blocks and $M_{i_f}(C)$ index blocks in memory for collection C, by using Eq. 3, we define the probabilities of the block containing the queried document and the block containing the index entry being in the cache ($P_d(C)$ and $P_{i_f}(C)$) as proportions of the total number of data and index blocks. In the case of hashed memory mapping, these proportions can be taken with the document or index sizes as there is no block structure in memory (detailed description in Sect. 3.2.1).

$$P_d(C) = \frac{M_d(C)}{B_d(C)} \qquad P_{i_f}(C) = \frac{M_{i_f}(C)}{B_{i_f}(C)} \tag{3}$$

[5] https://docs.mongodb.com/manual/core/distributed-queries

Next, we define the cost function for random access through an indexed field using the above equations. First, the relevant block containing the index of the document needs to be fetched. This block could reside in the cache with probability $P_{i_f}(C)$ or should be retrieved from disk with probability $1 - P_{i_f}(C)$. Next, the block containing the document needs to be retrieved, and this could be from the cache with a probability of $P_d(C)$ or the disk with a probability of $1 - P_d(C)$. Thus, the total cost is the average of retrieving the index and the document blocks as follows.

$$\begin{aligned} Cost_{Rand} = & \frac{T_m * P_{i_f}(C) + T_d * (1 - P_{i_f}(C))}{2} \\ & + \frac{T_m * P_d(C) + T_d * (1 - P_d(C))}{2} \end{aligned} \tag{4}$$

If the indexed field is that of a typical primary index of a B-tree, the index components can be omitted as the index is contained in the internal nodes (this is not the case in the current MongoDB, refer Sect. 4.2). Now, if we assume that the cost of reading a block from the cache can be neglected compared to the cost of reading from the disk ($T_m \ll T_d$), we can simplify the cost to $\frac{T_d * (2 - (P_{i_f}(C) + P_d(C)))}{2}$. Moreover, considering that block sizes are constant in the system, by replacing $P_{i_f}(C)$ and $P_d(C)$ from Eq. 3, we can infer that the cost of random access is negatively correlated with the size of the memory allocated to the index and data, and positively on the collection size (i.e., the size of the B-tree). The collection size is a product of the number of documents and the average document size divided by the fill factor (F in Table 1). We define the memory allocated to a particular collection as the sum of memory used by data and all the indexes (I).

$$M_{alloc}(C) = M_d(C) * Bsize_d + \sum_{k=0}^{I} M_{i_k}(C) * Bsize_i \tag{5}$$

Finally, our generic cost model uses memory usage as an external parameter specific to the underlying technology.

3.2 Specific Component

The specific component consists of three segments, namely, storage structure, memory mapping, and memory associativity. Both Couchbase Server and MongoDB use a B-tree structure to store data and indexes. Since the estimation of B-tree size is a familiar process [18,25], we focus on memory mapping and memory associativity in detail.

When the document store is started, the cache is assumed to be empty (i.e., cold start). Thus, all the requests sent involve fetching data from disk and caching them in memory. This will continue until the cache becomes full, and the cache eviction starts to release some of the blocks to allow new ones to be cached. As shown by previous work, the cache becomes stable in terms of the memory allocated to the different collections and indexes after a certain point, and its state can consequently be approximated [4, 10, 15, 31]. However, this approximation depends on the specific approaches used for managing the memory.

3.2.1 Memory Mapping

There are two forms of memory mapping that we explore in our work: direct and hashed. First, we define the unique queries issued in the workload $\mathbb{Q}$ as a set of triples. Each triple consists of a collection C, indexed field f, and the probability of using that indexed field $P(C, f)$.

Direct *(D)* In direct memory mapping, both data and index are stored, retrieved, and managed in memory as blocks. Thus, the formulas used from this point onwards apply to both the data B-tree and the (secondary) index B-tree. We define the number of repetitions of the indexed field f of a collection C as the ratio between the total number of leaf level entries and the number of distinct values of f as follows. When the value is a primary index or has a unique constraint, $Rep_f(C) = 1$ (which is used in Eq. 11).

$$Rep_f(C) = \frac{|C| * Mult_{i_f}(C)}{distinct(f)} \tag{6}$$

We assume that just before the eviction starts, there have been $|Q|$ issued queries and we define this state as the saturation point. These queries are from all the collections that are being accessed. Thus, each collection has $Req_f(C)$ number of document requests from each field f, which is proportional to its access frequency.

$$Req_f(C) = |Q| \cdot P(C, f) \tag{7}$$

However, the same document or the index can be requested more than once. Therefore, we estimate the number of unique requests $E_f(C)$ as the expected value after issuing $Req_f(C)$ requests out of the total number of distinct values with replacements.

$$E_f(C) = distinct(f) * \left(1 - \left(\frac{distinct(f) - 1}{distinct(f)}\right)^{Req_f(C)}\right) \tag{8}$$

Then, we define the selectivity factor in a collection C, with respect to a field f as $SF_f(C)$, which is the probability of a document being requested through the index on f. Using Eq. 8, we define this as $\frac{E_f(C)}{distinct(f)}$. However, there could be multiple queries that access the same collection through different indexes. Therefore, we aggregate the selectivity factor of a collection by using the formula for the probability of union on n events as follows. The number of all the queries issued on the document store is denoted by $|\mathbb{Q}|$.

$$SF(C) = \sum_{i=1}^{|\mathbb{Q}|}(-1)^{i+1} \Big(\sum_{1 \leqslant < k_1 \ldots < k_i \leqslant |\mathbb{Q}|} (SF_{f_{k_1}}(C) \wedge \ldots \wedge SF_{f_{k_i}}(C)) \Big) \tag{9}$$

If a data block is in the cache, at least one of the documents in the data block must have been requested. So, the probability of a document not being requested is $1-SF(C)$, and the probability of none of the documents in a data block being requested is $(1 - SF(C))^{R_d(C)}$. Hence, the probability of a data block being requested by a query $P_d^{req}(C)$ is the complement of none of its documents being requested by that query. In turn, the index B-tree has the same $SF(C)$ as it needs to be accessed in order to access the document. Even though the selectivity factor of the secondary index and the data B-tree is the same, there is one crucial difference between the physical storage of the two structures. The secondary index B-tree is sorted by the indexed value while the data B-tree is not. On account of this, the probability of a leaf node of the secondary index not being requested is that of none of the unique index values within the index block being requested. The index block contains $R_{i_f}(C)$ index entries and the number of unique values within the block is $\frac{R_{i_f}(C)}{Rep_f(C)}$.

$$P_d^{req}(C) = 1 - (1 - SF(C))^{R_d(C)} \tag{10}$$

$$P_{i_f}^{req}(C) = 1 - (1 - SF_f(C))^{\frac{R_{i_f}(C)}{Rep_f(C)}} \tag{11}$$

Consequently, at the saturation point, just before the eviction starts, the size of cached data $M_d^{sat}(C)$ and index $M_i(C)$ can be stated as follows.

$$M_d^{sat}(C) = B_d(C) * P_d^{req}(C) \tag{12}$$

$$M_{i_f}^{sat}(C) = B_i(C) * P_{i_f}^{req}(C) \tag{13}$$

Hashed *(H)* In a hashed memory mapping system, memory is managed per document. The document is brought into memory with its metadata. Here, when there is enough memory for all metadata, only documents are evicted while the metadata remains in memory. Since the relationship between the blocks and the documents are lost in hashing. Thus, we only estimate

the size of documents in the cache instead with Eqs. 14 and 15.

$$M_{i_f}(C) = Size_{i_f}(C) * |C| \tag{14}$$

$$M_d(C) = M_{alloc}(C) - Size_{i_f}(C) * |C| \tag{15}$$

In the case of not having enough memory to allocate all metadata, a full eviction mode could be used. In this case, the metadata is evicted when the document is evicted from memory. Here, the total memory used by a collection is divided proportionately to the size of the document and metadata, as in Eqs. 16 and 17.

$$M_d(C) = M_{alloc}(C) \cdot \frac{Size_d(C)}{Size_{i_f}(C) + Size_d(C)} \tag{16}$$

$$M_{i_f}(C) = M_{alloc}(C) \cdot \frac{Size_{i_f}(C))}{Size_{i_f}(C) + Size_d(C)} \tag{17}$$

With regard to Eq. 3, hashed memory mapping loses the block information. Thus, we use values from Eqs. 14 to 17 as the numerator and the size of the collection as the denominator (e.g. $\frac{R_d(C)}{F}$) to get the proportion of what is in memory out of the overall collection/index.

3.2.2 Memory Associativity

Memory associativity describes how memory is allocated between the collections. Here, we allocate a constant overhead K as extra memory used for parameters that are not considered in the formula. For example, it could be the internal nodes of the B-trees. Moreover, not all the memory is used by the collections and indexes and an upper memory limit is set. The eviction takes place once this limit is reached. We introduce this upper limit as a percentage denoted by u. In **predetermined** (P) associativity the memory is decided by the user.

$$M_{alloc}(C) + K = M_{user}(C) \tag{18}$$

In **shared** (S) associativity, the overall memory is shared between different collections which can be formalized as follows.

$$\sum_{i=1}^{N} M_{alloc}(C_i) + K = uM \tag{19}$$

3.2.3 Cache Eviction Policy

A Cache and its eviction policy is applicable when there is shared memory associativity. We introduce the formulas for a B-tree based, **LRU** (L) cache eviction policy as it is considered to be fair in most of the use cases.

When eviction cycles start, the least recently used blocks are removed from memory. Suppose a document (resp. index entry) is accessed with a probability P_{doc}. In that case, the likelihood of a leaf block in the data B-tree (resp. index B-tree) being accessed is the probability that some of the documents in that block are requested, which is $1-(1-P_{doc})^{R_d(C)}$, noted as P_d^{block}. To evict an internal block in the data B-tree (resp. index B-tree), all the leaf blocks pointed by that internal block need to be evicted. Hence, the probability of one of the leaf blocks not being referred is $1 - P_d^{block}$, and consequently the probability of some of these leaf blocks is referred is $1-(1-P_d^{block})^{R_{int}(C)}$ noted as P_d^{inter}. Since they depend one on another and $R_{int}(C) \gg R_d(C)$, it is clear that $P_d^{inter} \gg P_d^{block}$, and we can safely assume that the internal data blocks are hardly evicted (only in extreme cases). The same reasoning can be done for an index B-tree (notice that the probability of accessing a document is the same as the probability of accessing its corresponding index entry, so we would similarly obtain P_i^{block} and P_i^{inter}). Therefore, for the sake of simplicity, we only consider the eviction of leaf nodes and assume that all the internal nodes of the data and the index B-trees are pinned to the cache and take constant K memory as explained above. Their eviction will only become significant when there is a substantial number of blocks in the leaves. If so, refer to Appendix A for a detailed calculation and extension of Eq. 10 to include the eviction of internal B-tree blocks.

We use the term reference entry to name an entry of an internal block which points to a leaf block, and define the average number of reference entries in an internal block $R_{int}(C)$, in terms of internal block size $Bsize_{int}(C)$, reference entry size $Size_{int}(C)$ and the corresponding fill factor. For index B-trees, the reference entry size depends on the field f.

$$R_{int}(C) = F \cdot \left\lfloor \frac{Bsize_{int}}{Size_{int}(C)} \right\rfloor \tag{20}$$

Thus, the value of K can be easily obtained by iteratively moving up on the B-trees, starting from the leaves and calculating the number of blocks at each level by dividing the previous by $R_d(C)$ (or $R_i(C)$ or $Bsize_{int}(C)$ depending on the kind of B-tree and level). Then, by solving the system of Eqs. 7–13 under the condition that the sum of memory used equals the total memory available as shown by Eq. 19, we obtain the memory distribution just before the eviction, $M_d^{sat}(C)$ and $M_{i_f}^{sat}(C)$.

When the cache is stable, the probability of bringing in a new data block of a collection $P_d^{in}(C)$ should be equal to the probability of evicting a data block from the same collection $P_d^{out}(C)$ and the same can be applied for the index B-tree. Thus, solving the following system of equations, together with Eq. 19 we can obtain the stable state of the memory.

$$\forall C_j : P_d^{in}(C_j) = P_d^{out}(C_j), \quad \forall f \in C_j : P_{i_f}^{in}(C_j) = P_{i_f}^{out}(C_j) \tag{21}$$

We define $Shots_d(C)$ as the number of queried data blocks at a given time window for a collection at the stable state. Among these queried blocks, $Shots_d^{in}(C)$ number of blocks are already residing in the memory and $Shots_d^{out}(C)$ blocks need to be fetched from the disk into the memory. Thus, the number of queries whose documents are found in the cache is proportional to the number of blocks already in the cache and the ratio of cached blocks.

$$Shots_d^{in}(C) = M_d^{sat}(C) \cdot \frac{M_d(C)}{B_d(C)} \tag{22}$$

$$Shots_{i_f}^{in}(C) = M_{i_f}^{sat}(C) \cdot \frac{M_{i_f}(C)}{B_{i_f}(C)} \tag{23}$$

The evictable data blocks of a collection $E_d(C)$ are those blocks that have not been accessed in the last eviction cycle (i.e., those least recently used).

$$E_d(C) = M_d(C) - Shots_d^{in}(C) \tag{24}$$

Therefore, the evictability of a certain block in memory is $\frac{E_d(C)}{M_d(C)}$. Now, we can define the probability of evicting a data block from a collection (similarly for the index) as being a weighted average as in Eq. 25.

$$P_d^{out}(C) = \frac{W_d(C) \cdot \frac{E_d(C)}{M_d(C)}}{\sum_{j=1}^{N} \left(W_d(C_j) \cdot \frac{E_d(C_j)}{M_d(C_j)} + \sum_{j=1}^{N} \left(W_i(C_j) \cdot \frac{E_{i_f}(C_j)}{M_{i_f}(C_j)} \right) \right)} \tag{25}$$

We introduce the weight $W_d(C)$ mainly due to the implementation specifics of the underlying document stores. For an ideal LRU cache eviction policy system where it can determine the exact least recently used blocks to be evicted, the value should be 1. Since tracking all the blocks in memory is expensive, different document store implementations enforce approximations of the least recently used blocks.

We define the probability of a block containing the requested document being in the cache as $P_d(C) = \frac{M_d(C)}{B_d(C)}$. Thus, we define the probability of bringing a new block of a collection to the cache with regard to all the collections that are being used.

$$P_d^{in}(C) = \frac{M_d^{sat}(C) \cdot (1 - P_d(C))}{\sum_{j=1}^{N} \left(M_d^{sat}(C_j) \cdot (1 - P_d(C_j)) + \left(\sum_{f=1}^{I} M_{i_f}^{sat}(C_j) \cdot (1 - P_{i_f}(C_j)) \right) \right)} \tag{26}$$

4 Applying the cost model

We introduced the generic and the specific cost model components in the previous section. Depending on the document store, the relevant specific component formulas can be used to determine the memory distribution. However, each document store can have its own implementation decisions that need to be taken into account. In this section, we take two document store implementations in detail and discuss how to apply the formulas introduced above.

4.1 Couchbase Server (*THP*)

Couchbase Server is a distributed multi-modal data store that provides scalability, low latency, and high throughput for key-value and JSON document storage. Couchbase Server manages data using buckets, which are a logical grouping of physical resources. It offers two types of buckets, namely Memcached and Couchbase, but we focus our work on the latter because it stores data both in memory and on disk (Memcached only uses memory).

The documents in the disk are stored in a B-tree structure (T). Buckets operate on these documents only when loaded into memory. If the requested document is not currently in memory, it is automatically brought in from the disk individually together with its metadata as a hash (H). A bucket has a quota of dedicated memory which is configured at creation time (P). When a bucket reaches 85% of the allocated memory, an item is evicted. Each document stored in Couchbase Server has a fixed metadata size (i.e., 56 bytes). If a document is being used, its metadata and id need to be in memory. By default, Couchbase recommends all the metadata to be in memory. In this scenario we can apply Eqs. 14 and 18 with $u = 0.85$.[6] However, this requires more memory when the number of documents grows. Fig. 2 shows the memory is allocated to data and metadata with the two different eviction approaches. Eviction of metadata is supported only from version 3.2 onwards. With default eviction policy, all metadata entries (i.e., $|C|$) will always be in memory, and x documents will use the rest. When evicting metadata is enabled, there will be y documents and the corresponding y metadata entries in memory ($x \leq y$). The metadata is evicted together with the document. Thus, we can apply Eqs. 16, 17 and 18 with $u = 0.85$.

Fig. 3 shows the distribution of the memory quota among metadata and the documents in five different

[6] https://docs.couchbase.com/server/5.1/architecture/db-engine-architecture.html (High water mark)

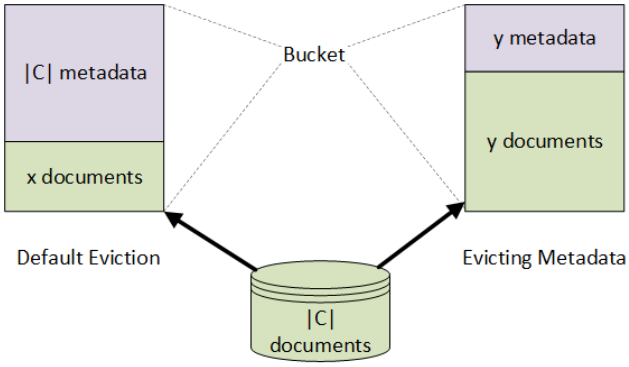


Fig. 2: Couchbase Server bucket usage

buckets with the same memory quota but different document sizes. Each of the buckets' average document size increases by a factor, but documents in all the buckets have the same index entry size. Therefore, the metadata size per document is the same. The chart shows that the memory ratio between data and metadata is affected by the document size. When the document size grows, few documents fit in the memory. Since the metadata in memory is only those of the documents also in memory, fewer metadata entries are leading to smaller memory usage.

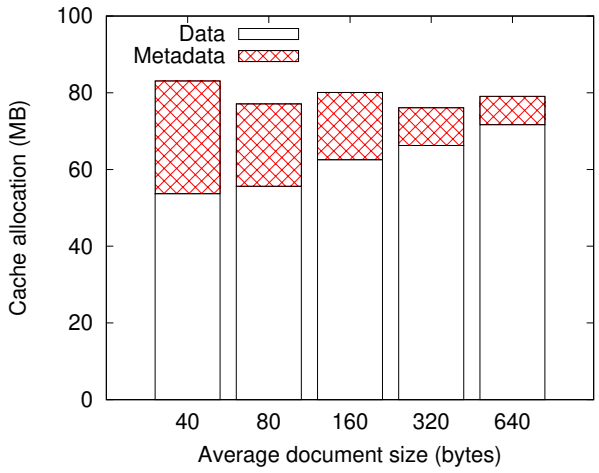


Fig. 3: Memory utilization in Couchbase Server

4.2 MongoDB (*TDSL*)

MongoDB stores data in BSON (binary JSON) format and supports ad-hoc queries such as field, range, regular expressions, and aggregation. The documents are stored in collections, have a primary identifier, and also support secondary indexes. It has a pluggable architecture where the end-user can select which storage engine to use. At the moment of writing, there are three main engines: MMAPv1, WiredTiger, and in-memory storage. From now on, we focus on WiredTiger as it is the default and more complex one.

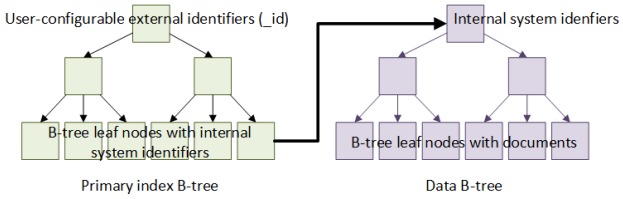


Fig. 4: MongoDB B-tree usage for primary key

The storage structure of WiredTiger is a B-tree or LSM-tree with a B-tree memory structure (T). However, as of MongoDB 4.2, only the B-tree storage structure is being used, and the LSM structure is not configurable. Moreover, in the current implementation of MongoDB, because of backward compatibility, the internal nodes of the B-tree do not contain the user-configurable external identifier (_id). Instead, as shown in Fig. 4, the documents are stored in a B-tree (*data B-tree* from now on) indexed by an internal system identifier. Then, there is a second B-tree (*index B-tree* from now on) where the leaf nodes contain the system identifiers of the data B-tree indexed by the user-configurable external identifier. Thus, the _id field behaves similar to a typical secondary index. The size of leaves is not fixed but capped with a maximum. All the collections and their indexes share a pre-defined cache memory zone (S), where all documents are brought in blocks (D). When the cache is full, the blocks are evicted to leave room for new blocks to be brought in. WiredTiger uses an LRU-like cache eviction policy (L) to evict underused blocks. Since the index and the documents are in two different B-trees, they behave independently in the WiredTiger cache eviction policy.

We carried out different experiments on a single MongoDB instance, randomly accessing documents from various collections changing different parameters. However, tests revealed inconsistencies concerning MongoDB specification. In particular, we identified that the cache eviction policy implementation was surprisingly prioritizing the eviction based on the collection's name. This is shown in Fig. 5, which depicts memory allocation for five identical collections with the same access frequency, being collection name the only difference (the average cache distribution is measured

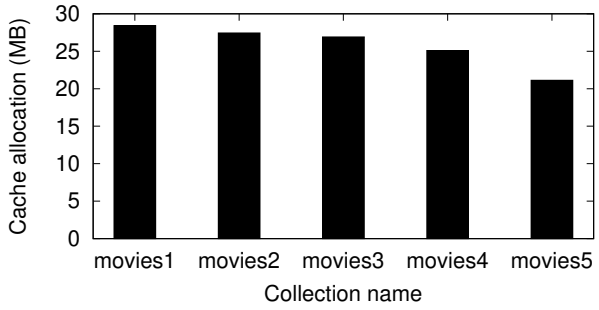


Fig. 5: MongoDB cache policy prioritizing the name

after 50,000 queries). The authors informed MongoDB about this issue and proposed a bug fix.[7]

Once the bug was solved (fixed in WiredTiger Release 3.2.1), we found three factors that affect the distribution of the cache among the collections and their indexes, namely access frequency, average document size, and the number of documents. Fig. 6 shows the distribution of the cache among different collections and their primary indexes after several queries, once the cache is full and stabilized after several eviction cycles (we capped the memory of MongoDB to 256 MB, issued 50,000 random access queries on different collections and took the measures by reading the cache metadata of every collection at the end).

Figs. 6a–6c show the effects of the access frequency, document size, and count on the cache distribution, respectively. It is visible that the frequency of access affects the distribution of the cache the most (as expected), while the impact of the document size and count is smaller. As shown in Fig. 6b, the memory allocated to the index decreases compared to that allocated to data when the documents get larger. On the contrary, when the document count changes, the memory used by the index increases while the memory of data drops as depicted by Fig. 6c.

We can apply Eqs. 7–13 under Eq. 19 with $u = 0.80$[8] for the saturation of the memory and Eqs. 21–26 together with Eq. 27 under Eq. 19 for the eviction in MongoDB. Yet, MongoDB only keeps track of 300 pages as eviction candidates in a queue. Each B-tree in use is walked to fill out this queue. The number of pages picked to fill this queue from a B-tree is proportional to the current memory occupation of the tree. Hence, $W_d(C)$ in Eq. 25 is proportional to the size of the memory occupation. A running example of applying these equations is presented in Appendix B.

$$W_d(C) = \frac{picks_d(C)}{queue\ size}$$
$$picks_d(C) = queue\ size \cdot \frac{M_d(C) \cdot Bsize_d}{M}$$
$$\therefore W_d(C) = \frac{M_d(C) \cdot Bsize_d}{M} \tag{27}$$

Other document stores can be included in our cost model with an analysis of their specific design decisions. For example, RethinkDB is a *TDSL* system similar to the one of MongoDB.[9]

5 Experiments

In this section, we validate our cost model through experiments with Couchbase Server and MongoDB. All experiments were carried out on a single node with Intel Xeon E5520, 24 GB of RAM running on Debian 4.9. Couchbase Server Community Edition version 5.1.1 was used with 1 GB dedicated to all the buckets. We used MongoDB Community Edition version 4.2, already modified to fix the bug explained above[7]. We also disabled the parallel execution of the eviction policy to obtain more stable results with fewer repetitions of the experiments. All experiments were conducted using MongoDB Java driver 3.8.2 and Couchbase Java client version 2.6.2. We conducted the experiments with hot cache for both Couchbase Server and MongoDB varying the frequency of access for MongoDB, bucket memory quota for Couchbase Server, average document size, and the number of documents for both. We generated synthetic data with flat documents for our experiments.[10] Despite nesting being relevant in evaluating different designs (it would generate different document sizes, access patterns and frequencies), it does not change the cost model itself, but only its parametrization. We used GEKKO Optimization Suite [1] to solve the systems of equations for cache distribution.

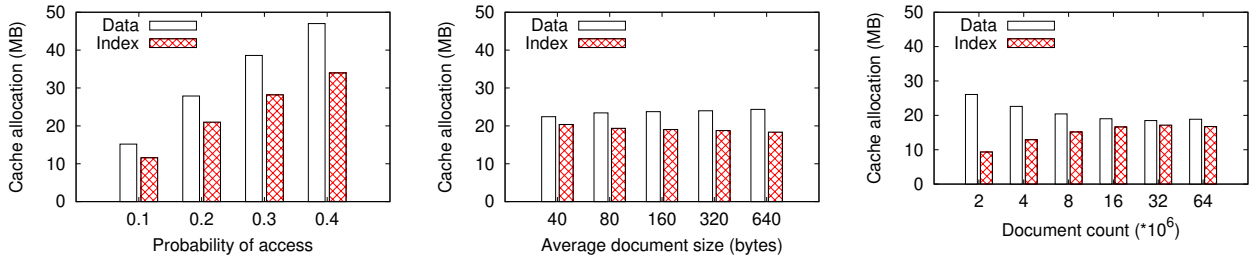
We measured the runtime individually for 50,000 random access queries (accessing documents through an index) in nanoseconds after the memory became stable for each of the experiments and took the average. Then, using the Pearson correlation coefficient, we measured how our estimates are related to the actual runtime values. However, the query cost estimation formulas introduced in Sect. 3 produce values without any unit. The actual runtime requires a multiplication factor which depends on external factors (i.e., hardware, operating

[7] https://jira.mongodb.org/browse/WT-4732

[8] http://source.wiredtiger.com/3.2.1/tune_cache.html (eviction_target)

[9] https://rethinkdb.com/

[10] The data generation and the experimental setup can be found in https://github.com/modithah/MongoExperiments



(a) Access frequency (b) Document size (c) Document count

Fig. 6: Effect of different parameters on cache distribution in MongoDB

system). Thus, to compare and plot the unitless estimated cost against the actual run time, we first do a min-max normalization on the two series separately. Then, the normalized series are shown in the same line chart. This is a common approach used to compare incomparable data by making them dimensionless [29]. The schema used for all of the experiments is shown in Listing 1. We used two integer fields for the primary (_id) and secondary index (s_index) fields. The range of the s_index values is used to change the repetitions with regard to Eq. 6, and the load field is used to adjust the size of the document depending on the experiment.

Listing 1: Schema used for experiments

```
{
  "_id": <int>,
  "s_index": <int{range}>,
  "load": <String(n)>
}
```

5.1 Couchbase Server

The retrieval of the documents through the primary index is done using the java client's `bucket.get({randomid})` command. As discussed in Sect. 4, Couchbase Server has fix-sized memory quota per bucket. Therefore, the access frequency does not affect the memory distribution. According to Eqs. 1, 2, 16, and 17 the memory distribution within a bucket depends on the average size of the documents. Fig. 7a compares our estimate of the memory distribution within a bucket using Eqs. 16 and 17 to the actual one. It is visible that the memory used by the data increases as the size of the documents grows.

Next, we used the memory distribution values in our cost model in Eqs. 3 and 4. We changed the size of the bucket and the average size of the stored documents and measured the average runtime for queries with random access through the primary index. Fig. 7b plots the estimated cost against the actual run time for different bucket sizes. Our estimation shows that there is a linear decrement of the run time when the bucket size is increased. This is also visible through the trend obtained by the actual runtime values. As shown in Fig. 7c, the runtime gradually increases with the size of the documents.

5.2 MongoDB

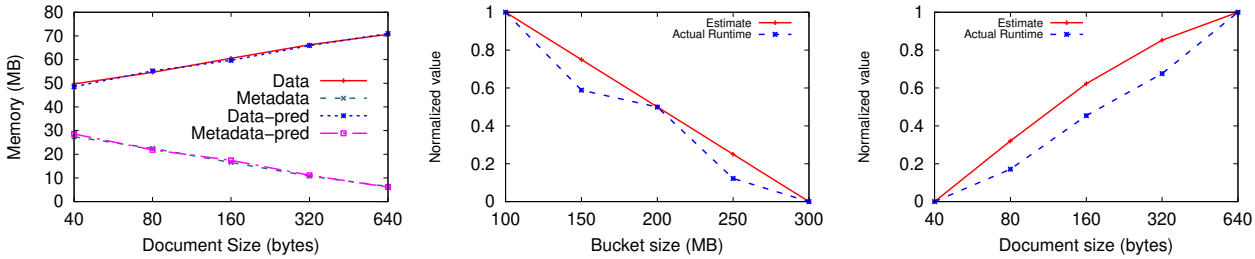
Our formulas for predicting memory distribution in MongoDB involve estimating two key factors:

a) The number of queries required to saturate the cache (by solving the system of equations Eqs. 7 to 13 under Eq. 19)
b) The distribution of the cache among different collections and indexes (by solving the system of equations Eqs. 21 to 26 together with Eq. 27 under Eq. 19 replacing $|Q|$ from saturation formulas).

For all of the experiments, we executed 50,000 random access queries, measured the cache distribution after every 100 queries, and obtained the average of 10 runs (the system was restarted after each run to reset the cache). We had to measure after every 100 queries because more frequent cache status requests affected the cache policy and the final memory distribution. We scrutinize the values of accessing a single collection and two collections. We varied the number of documents N, average document size $Size_d(C)$, frequency of accessing a collection $P(C, q)$, and the repetitions of the indexed value $Rep_v(C)$ as parameters of concern.

For a single collection, we included four tests:

Test 1 Fix the number of documents (13 million) and repetitions (1) while changing the average document size.
Test 2 Fix the average document size (80 B) and repetitions (1) while changing the number of documents.
Test 3 Fix the overall collection size and repetitions (1) while changing both document size and count at the same time.

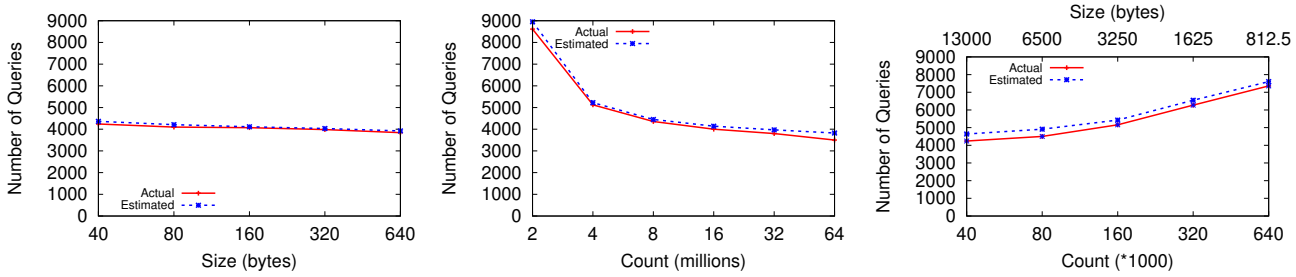


(a) Couchbase cache usage prediction

(b) Time estimate with different bucket size

(c) Time estimate with different document size

Fig. 7: Estimating the memory and time estimation in Couchbase Server



(a) Document size

(b) Document count

(c) Size and count

Fig. 8: Predicting saturation for a single collection with different parameters in MongoDB

Test 4 Fix both the average document size (80 B) and the number of documents (13 million) while changing the repetitions (secondary index).

For two collections, we conducted three other tests:

Test 5 Fix the document count (13 million), repetitions (1), and the frequency (50%) while changing the average size of the documents.

Test 6 Fix the average document size (320 B), repetitions (1), and the frequency (50%) while changing the number of documents.

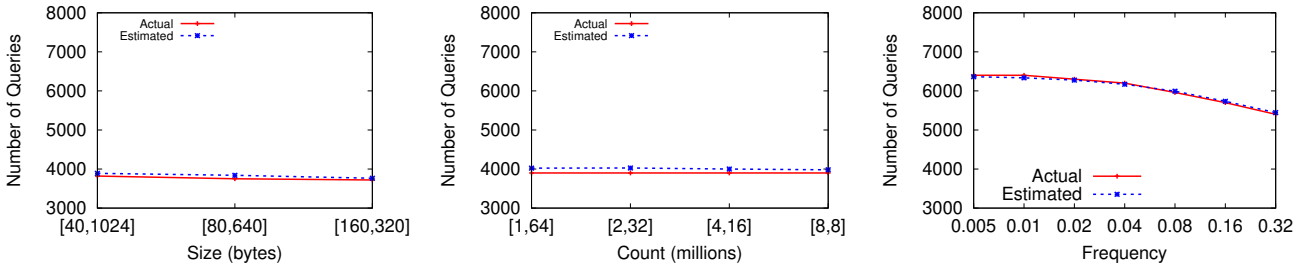
Test 7 Fix the average document size (1 kB), repetitions (1), and the number of documents (1 million) while changing the frequency.

Using the java client we issued `collection.findOne(new BasicDBObject("_id", {random id}))` for tests using primary indexes (Tests 1, 2, 3, 5, 6, and 7) and `collection.findOne( new BasicDBObject ("s_index", {randomvalue}))` for Test 4 involving the secondary indexes.

Predicting Saturation We used Eqs. 7–13 under Eq. 19 to estimate the saturation point ($|Q|$) and compared it with the average number of queries (different runs) before eviction starts.

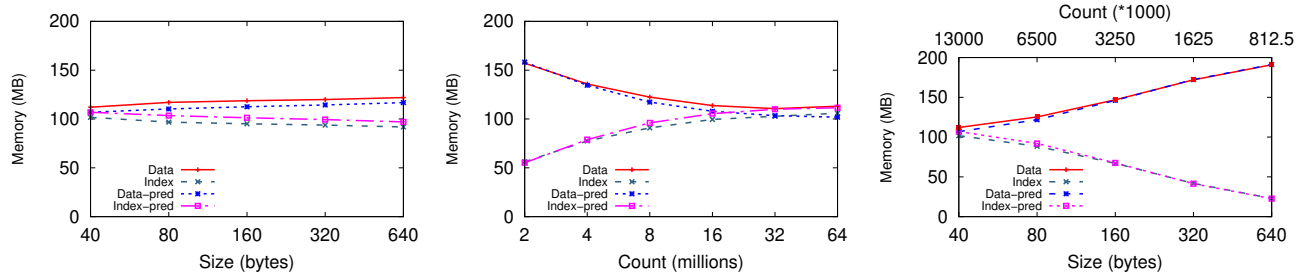
Fig. 8 illustrates the behavior of the saturation point of a single collection. Fig. 8a demonstrates that the saturation point is almost constant, with a slight decrease when the size of the documents grow on conducting Test 1. This is because the documents are accessed in blocks, and before saturation, there are many cache misses leading to bringing new blocks into memory. The number of requests remains almost constant because the probability of a miss is close to one in all cases, given the huge number of documents being used. As shown in Fig. 8b, with Test 2, it takes fewer queries to saturate the cache when the number of documents grows. This is due to both index and data B-trees being bigger with the higher number of documents, leading to fewer cache hits and resulting in fewer queries needed to saturate the cache. The impact of the document count is more significant than that of the document size as depicted in Fig. 8c, which shows that more queries are needed to saturate as the number of documents grows, and the document size shrinks in Test 3. This is because, the smaller the number of documents in the collection, the higher the hit rate, consequently higher the number of queries required.

We can also see how many queries are required to saturate the cache when accessing two collections (Fig. 9). The result of Test 5 is shown in Fig. 9a, where we clearly see that a few more queries are needed to saturate the cache (there is a slight downward trend) when the document size difference is higher to saturate



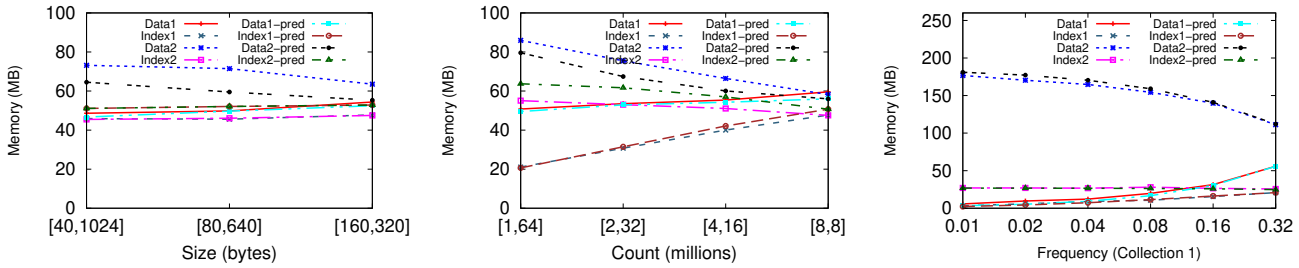
(a) Document size (b) Document count (c) Access frequency

Fig. 9: Predicting saturation for two collections with different parameters for MongoDB



(a) Document size (b) Document count (c) Size and count

Fig. 10: Predicting cache distribution for a single collection with different parameters in MongoDB

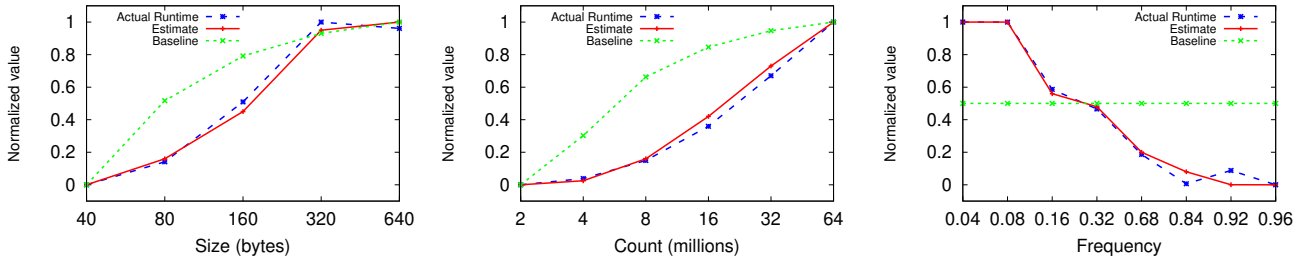


(a) Document size (b) Document count (c) Access frequency

Fig. 11: Predicting cache distribution for two collections with different parameters in MongoDB

the cache due to the higher hit rate of smaller document sizes. The saturation point is a bit lower than of the single collection, because of the space taken by the internal nodes pushing the memory to fill earlier (i.e., K is bigger). We noticed that the effect of the document size is more evident when there is a smaller number of documents (1 million). The effect of the document count (Test 6) is also negligible, demonstrated through Fig. 9b, whose values are comparable to the ones of single collections (Fig. 8b) beyond 16 million documents (notice that the sum of both collections is always above that). The only remarkable difference is that the saturation point is lower than that of a single collection due to more internal nodes being pinned. Finally, Fig. 9c shows the results of the saturation point of Test 7. We can see that more queries are needed to saturate the cache when the access frequency is low. The real reason, however, is that there is an opposite collection which is accessed with complementary frequency for each of the points (e.g., 0.995 for 0.005). The opposite collection has obviously more documents in the cache due to the higher access frequency leading to higher hit rates, and as a result, more queries are required to saturate the cache. Thus, the more balanced the frequencies of collections, the fewer queries are needed to saturate the cache.

Predicting the Cache Distribution Once we know when the memory saturates and starts being stable, we can analyze how memory is distributed among collections and indexes.



(a) Document size (b) Document count (c) Access frequency

Fig. 12: Time estimation comparison for different parameters in MongoDB

The outcome of the memory distribution for Test 1 is shown in Fig. 10a. When increasing the document size, the amount of memory devoted to data blocks grows while the memory for index blocks decreases. This happens because the data B-tree becomes larger with larger document sizes, and it occupies more memory, but the index B-tree size remains the same. However, the effect is minimal. Fig. 10b shows the results of Test 2 in measuring the effect of the number of documents on the memory distribution. The size of both B-trees grows with the number of documents. However, the index B-tree grows slower than the data B-tree, resulting in higher hit rates and more memory. The internal blocks of the B-tree increases as the number of documents increase. These internal blocks could also be getting evicted in the experiments whereas we assume them to be pinned in the cache. Thus, our estimation error gets higher as the number of documents increase. When we change both the size and the number of documents, keeping constant the collection size as per Test 3, we observe the trend augmented as shown in Fig. 10c (note that the axis of the number of documents is reversed due to the inverse relationship between the document count and the document size as we try to maintain the same overall collection size), because both factors favor the growth of memory used for the data (i.e., the less and bigger documents the more memory devoted to data and the less to index). When the repetitions increase in Test 4, a single index entry points to multiple documents, thus increasing the memory used by the data and decreasing the index. Our cost model is able to predict the memory distribution with a maximum error of 4%.

Fig. 11 illustrates in the corresponding subfigures the effect of the document size, number of documents, and the access frequency when accessing two collections. The gap between the memory used by the data decreases when the average document sizes are closer in Test 5, while the memory used by the index remains constant (Fig. 11a). Fewer documents fit in memory when the documents are larger, resulting in more cache misses and more data blocks need to be brought into the cache. When looking at Fig. 11b (Test 6), the memory used by the index of the first collection (*index-1*) increases while the one for data of the second collection (*data-2*) decreases when the difference between the number of documents of the two collections decreases. The memory used by *data-1* slightly increases and *index-2* decreases and align with *data-2* and *index-1* when the counts become identical. When a collection has more documents, there are more pinned internal nodes, and there are more cache misses requiring to bring more data and index blocks into the memory. When the collections are identical, the memory usage is similar. With regard to Fig. 11c (Test 7), the memory used by both data and index increases with the access frequency. When a collection is accessed with a higher rate, the evictability of a block becomes lower, resulting in more blocks residing in memory.

Query Cost Estimation Finally, once we have the memory distribution, we can analyze the performance of the system in a stable state using the generic cost functions. As illustrated by Fig. 12a (Test 1), the runtime increases as the size of the documents increases. Since the memory size is fixed, the number of documents that fit in the cache gets smaller resulting in more cache misses, leading to fetching more documents from the disk and increasing the runtime. As shown in Fig. 12b (Test 2), in the case of fixed-document size increasing the document count, the probability of a cache hit is higher on smaller document counts, and the runtime rises with the number of documents. The runtime gets lower as the frequency of access gets higher, as demonstrated by Fig. 12c (Test 7). This is because the collections with higher access frequency have more memory, which creates higher hit rates and lower runtimes. We calculated the miss rates of the hieracical cost model of Manegold et al. [20] and plotted the estimated costs as a baseline in Figs. 12a and 12b for

different document sizes and counts, respectively. We also include a horizontal line at 0.5 in the approximation in Fig 12c as all the cost estimations of Manegold et al. coincide in this case (an in depth discussion of that approach can be found in Sect. 5.4).

5.3 Accuracy of Prediction

Regarding our memory predictions, on looking at Fig. 7a, we see that our estimated trend is identical to the actual memory distribution of Couchbase Server with an average error of 3%. For MongoDB, this is a bit more complex, since we need the number of queries required to saturate the memory (Figs. 8 and 9), which is in general slightly overestimated, for an average error of 3% and a correlation of 0.995. By using it in solving the corresponding system of equations, we can predict memory usage of the data and indexes and accurately find the trend in all the cases with an under-estimation of the data while over-estimating the index, for an average error in all the scenarios of 6% for index and 5% for the data. As shown in Figs. 10a–11c, the estimates of the memory usage are highly correlated (0.995) with the actual values. With regard to Figs. 11a and 11b, prediction for *index-1* and *data-1* are almost perfect, but we underestimate *data-2* and overestimate *index-2*. The highest error we encountered is with the prediction of the *data-2* when changing the document size. When looking into Fig. 11c, we can see that the error increases for the number of data pages when increasing the probability of accessing the collection.

Overall, we have successfully predicted the allocation of the memory with a maximum error in all the experiments of 13% and an average error of 9% for the different number of documents, average document sizes, probability of access, and available memory. However, when it comes to the runtime estimates, the effect of this error becomes negligible. Regarding runtime predictions, we manage to find the runtime trend in Couchbase Server with a high correlation (0.93) between all our estimates and the actual values (values from Figs. 7b and 7c). The correlation is even higher (0.94) for MongoDB (values from Figs. 12a–12c). There is a slight difference between our estimate and the actual value when the access frequency is very low and very high (Fig. 12c), because the measured runtime values are very close to each other in extreme cases (three milliseconds). Thus, our approach enables us to identify the overall effect of the design decisions on runtime. This runtime, together with other parameters such as storage space and heterogeneity of a collection, can be used to evaluate design alternatives.

5.4 Comparison to Other Approaches

To the best of our knowledge, this work is the first generic cost model for document stores. However, we can compare our cost estimations against a generic cost model for hierarchical memory systems (hereinafter referred as the hierarchical cost model) [20]. This cost model is based on in-memory database systems and can be extended to the disk I/O layer. However, unlike the database system discussed in the hierarchical cost model, which relies only on the operating system cache, document stores have their own cache management system. Our cost model has the capacity to handle these implementation-specific cache policies, and we have shown it by applying them on two different document stores, Couchbase Server and MongoDB.

Random access in the hierarchical cost model is equivalent to the random access queries discussed in this paper. Nevertheless, it requires the number of random accesses performed to model the cost, which is not required in our approach that only relies on the stability of the cache. Moreover, the Sterling numbers and the factorials used in the calculations quickly grow quite large making the calculations almost impossible for large numbers (the largest Sterling number the authors have used in their experiments is 1024 which corresponds to the number of L1 cache lines [19]) unless you make mathematical approximations on the formulas, thus increasing the error. Therefore, the approach used in the hierarchical cost model is not scalable to the experiments that we have carried out as database caches are larger than 1024 blocks. However, we substituted Eq. 8 for the expected value whose result for small values exactly coincide.

The concurrent execution formulas in the hierarchical cost model assume that the cache contains a fraction of data regions involved proportional to their footprint size (i.e., the size of the collection). However, our experimental results show otherwise, especially with different access frequencies (see Fig. 6) and eviction policies. The pattern is even more complex when it comes to secondary indexes where a sequential accesses on the index are followed by multiple random accesses to the documents. The comparison results in Figs. 12a–12c further confirm that a estimation specific for document stores, is clearly superior to a generic approach for caching.

6 Conclusions

Document stores have now become one of the most widely adopted NoSQL stores. However, the schema design concepts in them allows the end users to have multiple designs depending on their needs. These choices

are based on generic rules and guidelines [13], as opposed to a formal methodology like normalization in RDBMS. However, the lack of standards or cost models makes it challenging to determine which design would perform better than the others. Thus, we introduced a generalized cost model for document stores, with a pluggable component for storage structures and memory management decisions. The model is applicable to a B-tree based disk storage with hashed or direct memory mappings with pre-determined or shared memory associativity with LRU cache eviction policy.

We evaluated our formal model using MongoDB and Couchbase server as exemplars with different implementation details in the memory management under a fixed workload, and estimated memory usage with a minimum precision of 13% and an average precision of 91% overall. Nevertheless, the effect of this error on the final query cost is minimal, and our estimates are highly correlated (0.953 considering both systems) to the actual execution times. By using this same model, we also managed to find flaws in existing implementations and suggested the corresponding improvements, that have already been accepted by the corresponding communities of developers.

As future work, we plan to extend the cost model into native JSON storage in RDBMS. Through this, we will be able to evaluate the performance of having storage solutions in RDBMS, thus enabling data design techniques beyond normalization theory. Moreover, the cost model formulas can be further enhanced to other document stores with different memory and storage implementations. By having such a cost model, we can estimate the cost of a given design and query workload. We have already implemented DocDesign, a tool to determine the optimal schema design for document stores using the cost model discussed here together with other parameters such as storage requirements for a specific use case and workload [12]. Using approaches such as multi-criteria optimization together with this cost model, will facilitate the automation of the design rather than manually applying generic rules.

Acknowledgements This research has been funded by the European Commission through the Erasmus Mundus Joint Doctorate Information Technologies for Business Intelligence - Doctoral College (IT4BI-DC)

References

1. L. D. R. Beal, D. C. Hill, R. A. Martin, and J. D. Hedengren. GEKKO Optimization Suite. *Processes*, 6(8):106–131, 2018.
2. E. Bertino and P. Foscoli. On Modeling Cost Functions for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):500–508, 1997.
3. R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.
4. A. Dan and D. Towsley. An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes. *SIGMETRICS Performance Evaluation Review*, 18(1):143–152, 1990.
5. A. de la Vega, D. García-Saiz, C. Blanco, M. E. Zorrilla, and P. Sánchez. Mortadelo: Automatic generation of NoSQL stores from platform-independent data models. *Future Generation Computer Systems*, 105, 2020.
6. R. Fagin. Asymptotic Miss Ratios Over Independent References. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.
7. H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.
8. G. Gardarin, J. Gruser, and Z. Tang. A Cost Model for Clustered Object-Oriented Databases. In *International Conference on Very Large Data Bases*, pages 323–334, 1995.
9. G. Gou and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.
10. F. Guo and Y. Solihin. An Analytical Model for Cache Replacement Policy Performance. *SIGMETRICS Performance Evaluation Review*, 34(1):228–239, 2006.
11. R. Hecht and S. Jablonski. NoSQL Evaluation: A Use Case Oriented Survey. In *IEEE International Conference on Cloud and Service Computing*, pages 336–341, 2011.
12. M. Hewasinghage, A. Abelló, J. Varga, and E. Zimányi. DocDesign: Cost-Based Database Design for Document Stores. In *International Conference on Scientific and Statistical Database Management (SSDBM)*, 2020.
13. A. A. Imam, S. Basri, R. Ahmad, J. Watada, M. T. Gonzalez-Aparicio, and M. A. Almomani. Data Modeling Guidelines for NoSQL Document-Store Databases. *International Journal of Advanced Computer Science and Applications*, 9(10):544–555, 2018.
14. Y. E. Ioannidis. Query Optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.
15. B. Jiang, P. Nain, and D. Towsley. LRU Cache under Stationary Requests. *SIGMETRICS Performance Evaluation Review*, 45(2):24–26, 2017.
16. J. Kim, W. Lee, and K. Lee. The Cost Model for XML Documents in Relational Database Systems. In *IEEE International Conference on Computer Systems and Applications*, pages 185–187, 2001.
17. W. F. King III. Analysis of Demand Paging Algorithms. In *IFIP Congress (1)*, pages 485–490, 1971.
18. S. Lightstone, T. J. Teorey, and T. P. Nadeau. *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more*. Morgan Kaufmann, 2007.
19. S. Manegold, P. Boncz, and M. Kersten. Generic database cost models for hierarchical memory systems. *[INS]. CWI.*, (Technical Report INS-R0203), Jan. 2002.
20. S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *International Conference on Very Large Data Bases*, pages 191–202, 2002.
21. N. Megiddo and D. S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *IEEE Computer*, 37(4):58–65, 2004.

22. J. Michels, K. Hare, K. Kulkarni, C. Zuzarte, Z. H. Liu, B. Hammerschmidt, and F. Zemke. The New and Improved SQL: 2016 Standard. *ACM SIGMOD Record*, 47(2):51–60, 2018.
23. M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu. NoSE: Schema design for NoSQL applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(10), 2017.
24. M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems 4th ed.* Springer Science & Business Media, 2020.
25. R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Third edition, 2003.
26. J. Schindler. I/O Characteristics of NoSQL Databases. *Proceedings of the VLDB Endowment*, 5(12):2020–2021, 2012.
27. A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.
28. A. J. Smith. Cache Evaluation and the Impact of Workload Choice. *SIGARCH Computer Architecture News*, 13(3):64–73, 1985.
29. N. Vafaei, R. A. Ribeiro, and L. M. Camarinha-Matos. Data normalisation techniques in decision making: case study with TOPSIS method. *International Journal of Information and Decision Sciences*, 10(1):19–38, 2018.
30. T. Vajk, L. Deák, K. Fekete, and G. Mezei. Automatic NoSQL schema development: A case study. In *Artificial Intelligence and Applications*, 2013.
31. C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache Modeling and Optimization using Miniature Simulations. In *USENIX Annual Technical Conference*, pages 487–498, 2017.
32. J. Yao. An Efficient Storage Model of Tree-Like Structure in MongoDB. In *IEEE International Conference on Semantics, Knowledge and Grids*, pages 166–169, 2016.

A Calculating Internal B-tree Blocks

We calculated the probability of a leaf block of data B-tree in memory being requested $P_d^{req}(C)$ as $1-(1-SF(C))^{R_d(C)}$, in Eq. 10. . Hence, we continue the calculation of the internal nodes of the data B-tree as follows.

If a document is in the cache, the data block containing the document and the internal block containing the reference entry to that data block must be in the cache. On the contrary, if an internal block is not in the cache, none of the data blocks pointed by the reference entries in it can be in the cache. Therefore, for an internal block not to be in the cache, all of the reference entries of the block should reference blocks, not in the cache. Thus, since the probability of a single reference entry referring to a leaf block not in the cache is $1-P_d^{req}(C)$, and there are $R_{int}(C)$ reference entries in a single internal block, the probability of an internal block in memory being requested can be defined as follows.

$$P_{int}^{req}(C) = 1-(1-P_d^{req}(C))^{R_{int}(C)} \tag{28}$$

Moreover, we estimate the number of internal blocks pointing to the leaves, $Inter(C)$ as follows.

$$Inter(C) = \left\lceil \frac{\lceil \frac{|C|}{R_d(C)} \rceil}{R_{int}(C)} \right\rceil \tag{29}$$

Finally, we can state the cached internal blocks $M_{int}(C)$ as follows.

$$M_{int}(C) = Inter(C) * P_{int}^{req}(C) \tag{30}$$

B Cost Calculation Examples for MongoDB

We present the application of our cost model in MongoDB with two examples. First, a single collection accessed through primary index with complete set of equations and calculations. Second, we present a real world usecase with only the initial calculation of the inputs because the complexity and the number of equations increase in such scenario.

B.1 Single Collection with Primary Index

Let us take an scenario with Test 1 (13 million documents) and average document size of 40 bytes. First, we calculate the average number of documents and index entries in a block, together with the total number of data and index blocks as follows (by applying Eqs.1 and 2).

$$|C| = 13*10^6 \quad Bsize_d = 32Kb \quad Bsize_i = 32Kb$$
$$Size_d(C) = 40b \quad Size_{i\_id}(C) = 22b \quad F = 0.7$$
$$R_d(C) = 0.7 \cdot \left\lfloor \frac{32768}{40} \right\rfloor = 573$$
$$R_{\_id}(C) = 0.7 \cdot \left\lfloor \frac{32768}{22} \right\rfloor = 1042$$

$$B_d(C) = \left\lceil \frac{13*10^6}{573} \right\rceil = 22676 \quad K = 10Mb \quad M = 256Mb$$
$$B_{\_id}(C) = \left\lceil \frac{13*10^6*1}{1042} \right\rceil = 12473 \quad u = 0.80$$

Since we have only the primary index, $Rep_{\_id} = 1$, $P(C, \_id) = 0.5$, and $P(C) = 0.5$. Now, applying Eqs. 7–13 together with Eqs. 4 and 19, we come up with the following set of equations.

$$Req_{\_id}(C) = |Q| \cdot 0.5$$
$$E_{\_id}(C) = 13*10^6 * \left(1 - \left(\frac{13*10^6-1}{13*10^6}\right)^{Req_{\_id}(C)}\right)$$
$$SF_{\_id}(C) = \frac{E_{\_id}(C)}{13*10^6} = \left(1 - \left(\frac{13*10^6-1}{13*10^6}\right)^{Req_{\_id}(C)}\right)$$
$$SF(C) = SF_{\_id}(C) = \left(1 - \left(\frac{13*10^6-1}{13*10^6}\right)^{Req_{\_id}(C)}\right)$$
$$P_d^{req}(C) = 1-(1-SF(C))^{573}$$
$$P_{\_id}^{req}(C) = 1-(1-SF_{\_id}(C))^{1042}$$
$$M_d^{sat}(C) = 22676 * P_d^{req}(C) \quad M_{\_id}^{sat}(C) = 12473 * P_{\_id}^{req}(C)$$

$$M_d^{sat}(C)*32768 + M_{\_id}^{sat}(C)*32768 = ((0.8*256)-10)*1024^2$$

By solving the above set of equations, we obtain the values for $|Q| = 4242.85$, $M_d^{sat}(C) = 3038.98$, and $M_{\_id}^{sat}(C) = 2847.82$. Using these values at the memory saturation point, we can come up with the following set of equations by applying Eqs. 21–27 together with Eqs. 4 and 19.

$$P_d^{in}(C) = P_d^{out}(C) \quad P_{\_id}^{in}(C) = P_{\_id}^{out}(C)$$
$$Shots_d^{in}(C) = 3038.98 \cdot \frac{M_d(C)}{22676}$$
$$Shots_{\_id}^{in}(C) = 2847.82 \cdot \frac{M_{\_id}(C)}{24962}$$
$$E_d(C) = M_d(C) - Shots_d^{in}(C)$$
$$E_{\_id}(C) = M_{\_id}(C) - Shots_{\_id}^{in}(C)$$

$$W_d(C) = \frac{M_d(C) \cdot 32768}{((0.8 * 256) - 10) * 1024^2}$$
$$W_{\_id}(C) = \frac{M_{\_id}(C) \cdot 32768}{((0.8 * 256) - 10) * 1024^2}$$
$$P_d^{out}(C) = \frac{W_d(C) \cdot \frac{E_d(C)}{M_d(C)}}{W_d(C) \cdot \frac{E_d(C)}{M_d(C)} + W_{\_id}(C) \cdot \frac{E_{\_id}(C)}{M_{\_id}(C)}}$$
$$P_{\_id}^{out}(C) = \frac{W_{\_id}(C) \cdot \frac{E_{\_id}(C)}{M_{\_id}(C)}}{W_d(C) \cdot \frac{E_d(C)}{M_d(C)} + W_{\_id}(C) \cdot \frac{E_{\_id}(C)}{M_{\_id}(C)}}$$
$$P_d^{in}(C) = \frac{3038.98 \cdot (1 - 0.5)}{3038.98 \cdot (1 - 0.5) + 2847.82 \cdot (1 - 0.5)} = 0.52$$
$$P_{\_id}^{in}(C) = \frac{2847.82 \cdot (1 - 0.5)}{3038.98 \cdot (1 - 0.5) + 2847.82 \cdot (1 - 0.5)} = 0.48$$
$$M_d(C) * 32768 + M_{\_id}(C) \cdot 32768 = ((.8 * 256) - 10) * 1024^2$$

By solving the above equations we obtain $M_d(C) = 2847.82$ and $M_{\_id}(C) = 3038.98$. By applying these values on Eqs. 3 and 4 we get a relative cost for a query through $\_id$ as follows.

$$P_d(C) = \frac{2847.82}{22676} = 0.12 \quad P_d(C) = \frac{3038.98}{12473} = 0.24$$
$$Cost_{Rand} = 2 - (0.24 + 0.12) = 1.64$$

B.2 Multiple Collections

Let us take a use-case of storing the data of authors and their books. Let's assume that we chose to have a reference to the authors inside each of the books (as shown in Listing 2) out of the possible design choices. Moreover, let us also assume that each author has 5 books and each book has 3 authors on average.

Listing 2: Example schema of a document store

```
"Books":{
    "_id": <int>,
    "B_NAME": <varchar>,
    "Authors": [ "A_ID": <int> ]
},
"Authors":{
    "_id": <int>,
    "A_NAME": <varchar>
}
```

In this scenario, we have two collections and three indexes (two primary on each of the collections and one secondary index on A_ID in Books). We calculate the number of documents/indexes in a block and the total number of blocks for each of these five B-trees as follows.

$|\mathsf{Books}| = 4 * 10^6 \quad |\mathsf{Authors}| = 2.5 * 10^6 \quad Bsize_{d/i} = 32Kb$
$Size_d(\mathsf{Books}) = 265b \quad Size_d(\mathsf{Authors}) = 150b \quad F = 0.7$
$Size_i(\mathsf{Books/Authors}) = 22b \quad R_d(\mathsf{Authors}) = 152$
$R_d(\mathsf{Books}) = 89 \quad B_d(\mathsf{Authors}) = 16448$
$B_d(\mathsf{Books}) = 44944$
$R_{\_\mathsf{id}}(\mathsf{Books}) = R_{\mathsf{A\_ID}}(\mathsf{Books}) = R_{\_\mathsf{id}}(\mathsf{Authors}) = 1042$
$B_{\_\mathsf{id}}(\mathsf{Books}) = 3843 \quad B_{\_\mathsf{id}}(\mathsf{Authors}) = 2402$
$Mult_{\mathsf{A\_ID}}(\mathsf{Books}) = 5 \quad B_{\mathsf{A\_ID}}(\mathsf{Books}) = 19213$

The following queries are executed with equal probability (0.25) on our documents store.

Q1 Find the author name by _id
Q2 Find the book name by _id
Q3 Find all the book names with a given _id of an author
Q4 Find all the author names with a given _id of a book

Since our cost model depends on the access probability on each of the B-tree structures, we calculate them as shown in Table 2. In Q3, we have single access to the secondary index on A_ID and five access to the data B-tree. Q4 involves two queries, first, one to retrieve a book through its _id and then on average, there would be 3 A_IDs which need to be retrieved as three independent requests through _id of the Authors collection.

Table 2: Calculating the access probability of the B-trees

	Index usage			Collection usage	
	Book		Author	Book	Author
	_id	A_ID	_id		
Q1	-	-	0.25	-	0.25
Q2	0.25	-	-	0.25	-
Q3	-	0.25		5*0.25	-
Q4	0.25	-	3*0.25	0.25	0.75
Total	0.5	0.25	1	1.75	1
Probability	0.111	0.056	0.222	0.389	0.222

Now, we have the final input for our cost model, together with $Rep_{\mathsf{A\_ID}}(\mathsf{Books})$ by applying Eq. 6, as follows:

$P(\mathsf{Book}) = 0.389 \quad P(\mathsf{Book}, \mathsf{\_id}) = 0.111$
$P(\mathsf{Book}, \mathsf{A\_ID}) = 0.056$
$P(\mathsf{Author}) = 0.222 \quad P(\mathsf{Author}, \mathsf{\_id}) = 0.222$
$$Rep_{\mathsf{A\_ID}}(\mathsf{Books}) = \frac{5 * 2.5 * 10^6}{4 * 10^6} = 3.125$$

Now, we can apply Eqs. 7–13 together with Eqs. 4 and 19 on the inputs to obtain the values for memory distribution at the saturation point. Then, using these results on Eqs. 21–27 together with Eqs 4 and 19, we obtain the following final memory distribution. These calculations are similar to the example in Appendix B.1, but we omit listing them out due to their extensiveness.

$M_d(\mathsf{Books}) = 2532 \quad M_{\_\mathsf{id}}(\mathsf{Books}) = 638$
$M_{\mathsf{A\_ID}}(\mathsf{Books}) = 351$
$M_d(\mathsf{Authors}) = 1401 \quad M_{\_\mathsf{id}}(\mathsf{Authors}) = 935$

Finally, we calculate the miss rates and the relative cost of each of the queries as follows.

$$P_d(\mathsf{Books}) = \frac{2532}{44944} = 0.056 \quad P_{\_\mathsf{id}}(\mathsf{Books}) = \frac{638}{3843} = 0.166$$
$$P_{\mathsf{A\_ID}}(\mathsf{Books}) = \frac{351}{19213} = 0.018 \quad P_d(\mathsf{Authors}) = \frac{1401}{16448} = 0.08$$
$$P_{\_\mathsf{id}}(\mathsf{Authors}) = \frac{935}{2402} = 0.38$$
$Cost(Q1) = 2 - (0.38 + 0.08) = 1.54$
$Cost(Q2) = 2 - (0.166 + 0.056) = 1.778$
$Cost(Q3) = 1 - 0.018 + 5 * (1 - (0.056)) = 5.702$
$Cost(Q4) = 2 - (0.166 + 0.056) + 3 * (2 - (0.38 + 0.08))$
$= 6.398$