First Past the Post: Evaluating Query Optimization in MongoDB

Dawei Tao
University of Sydney
Sydney, Australia
dtao7193@uni.sydney.edu.au

Sidath Randeni Kadupitige
University of Sydney
Sydney, Australia
sidath.randenikadupitige@sydney.edu.au

Michael Cahill
MongoDB
Sydney, Australia
michael.cahill@mongodb.com

Alan Fekete
University of Sydney
Sydney, Australia
alan.fekete@sydney.edu.au

Uwe Röhm
University of Sydney
Sydney, Australia
uwe.roehm@sydney.edu.au

ABSTRACT

Query optimization is crucial for every database management system to enable fast execution of declarative queries. Most dbms use cost-based query optimization. However, MongoDB implements a different approach to choose an execution plan that we call *"first past the post" (FPTP) query optimization*. FPTP does not estimate costs for each execution plan, but rather it partially executes the alternative plans in a round-robin race, and it observes the work done by each, and how many records each has returned. The optimizer computes a score for each potential plan, based on the partial execution, and then it adopts the plan with highest score, to run to completion as the chosen plan for this query.

In this paper, we analyze the effectiveness of MongoDB's FPTP query optimizer. We see whether the optimizer will choose the truly best execution plan among all alternatives, and (if it does not) we find how much slower the chosen plan is, compared to the best plan for the query. We also show how to visualize the effectiveness, and in this way, we identify situations where the MongoDB 4.4.0 query optimizer chooses suboptimal query plans. Through experiments, we conclude that FPTP has preference bias, choosing an index scan plan even in many cases where a collection scan would run faster. We identify the reasons for the preference bias, which can lead to MongoDB choosing a plan with more than twice the runtime compared to the best possible plan for the query.

PVLDB Reference Format:
Dawei Tao, Sidath Randeni Kadupitige, Michael Cahill, Alan Fekete, and Uwe Röhm. First Past the Post: Evaluating Query Optimization in MongoDB. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at http://vldb.org/pvldb/format_vol14.html.

1 INTRODUCTION

Query optimization is a long-established topic in database management systems. For a given declarative query submitted by a user, there are many possible execution plans, each of which describes a correct way to calculate the result. The plans for a query can vary through orders of magnitude in cost, and so a query optimizer is vital, to choose a good (cheap to evaluate) plan among the possible ones. Most database management systems include an optimizer that is cost-based: it considers a variety of plans, estimates the cost of each plan from statistics, knowledge of the index structures etc, and it then chooses to execute the plan with lowest estimated cost among those it considered.

MongoDB [24] is a document-oriented database with rapidly growing popularity. Query optimization in MongoDB uses a very different approach which is not based on estimating costs of queries before they are run, but rather MongoDB runs many execution plans in a round-robin "race", allowing each to do a small amount of work at a time. After a point, it considers the outcomes so far, and calculates a score for each plan based on the work done and the number of results produced so far. The plan with the highest score wins the race, and it (alone) continues to be executed to completion as the chosen plan for this query. We call the MongoDB technique "first past the post" (FPTP) query optimization.

The central aim of our work is to evaluate and understand the current implementation of MongoDB's FPTP optimization. In this paper, we explain MongoDB's query optimizing technique in Section 2. We describe the innovative way we do the evaluation (and how we visualize the results) in Section 3. The results of this empirical study of MongoDB are in Section 4. We find that FPTP in MongoDB has a preference bias, it systematically avoids collection scan even when this is in fact a better plan. We explore the reasons for this in Section 5 then propose an improvement and evaluate its effectiveness.

This paper makes the following contributions:

(1) We describe in detail how the FPTP query optimizer in MongoDB chooses query plans.
(2) We propose an innovative way to evaluate and visualize the impact on query performance of an optimizer's choices. By using this approach, we identify places where the MongoDB query optimizer chooses sub-optimal query plans. Our approach could form the basis of an automated regression testing tool to verify that the query planner in MongoDB improves over time.
(3) We identify causes of the preference bias of FPTP, in which index scans are systematically chosen even when a collection scan would run faster.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

2 BACKGROUND AND RELATED WORK

There exists an extensive literature on query optimization, Here we point to some of the fundamental papers, and then we describe MongoDB (the platform we study in this paper) and how its FPTP query optimization is done.

2.1 Query Optimization Architecture

A vital factor in the spread of relational database technology was the support for declarative queries, in which a user expresses a query indicating *what* information is needed, and the system then finds out *how to* execute the query, that is, what steps to take that will retrieve that information from what is stored [11]. Declarative queries have remained important also with other data models, such as object and document data. In general, a given query will have many execution plans (the details will depend on the physical structures such as indices), and these plans will differ very much in performance. For declarative queries to be workable, the database management system must be able to optimize, that is, find an execution approach which is not only correct in retrieving the specified information, but also runs rapidly.

While a few early commercial systems used pure heuristics to choose the execution plan for a submitted query, the dominant approach is cost-based optimization, invented by Selinger and colleagues for System R [29]. The essential work of a cost-based optimizer is (i) generate a variety of *candidates*, which are execution plans each of which would calculate the correct results for the given query, (ii) estimate the cost each candidate plan will incur when it runs (iii) choose to actually execute the plan, among those considered, whose estimate cost is lowest. Cost-based optimization is now industry-standard practice [20].

Research has continued apace, both in industry and academia, to improve the design of query optimizers.

2.2 Candidate Plan Generation

For complicated queries, there are a very large number of execution plans that produce the correct results. The plans can vary logically (for example, in which order joins are done, whether selection happens before or after a join, etc) and physically (eg which index is used to find records, which join algorithm is used). The System R optimizer [29] restricted its attention to joins done in a particular set of sequential patterns. The generation of potential logical plans by successively applying transformations was introduced by Freytag [12] and Graefe [15] and used in IBM's Starburst project [17, 26]. Later Graefe extended this to a unified view where logical and physical choices were treated together in generating potential plans [14].

2.3 Cost Estimation

The cost one estimates is a measure of the resources consumed in calculation; traditionally, for a disk-based database management system, the dominant cost is bringing pages from disk to memory or vice-versa. To estimate the cost of an execution plan, one proceeds from estimates of the cost of each operation (eg join, select) within the plan. The cost of a step depends on the physical structure, but especially on the cardinality of the inputs and outputs of the step. For base collections, cardinality is clear, but subsequent operations take as inputs the results of prior calculations, and so estimating the cardinality requires estimating the selectivity of predicates, the number of distinct values in an attribute, the probability that items will match in a join, etc. Many techniques have been proposed for estimating specific operations such as projection, range, join, aggregation [3, 18, 22, 27]. Estimates are often based on statistics kept in the database, or calculated dynamically through sampling [25].

The cost estimate for a plan is very sensitive to errors in the estimates of cardinality, and so several researchers have tried to achieve *robust* planning, where the plan choice works well even if the estimates are mistaken. One approach is to include a measure of uncertainty in each estimate [4], and then choose a plan which is good throughout the likely range of estimates. Another is to check while the chosen plan is executing, and to stop and re-optimize if the reality differs too much from what was estimated [23]. These ideas have been combined [5].

2.4 Optimizer Implementation and Use

Graefe and others have emphasised the importance of modularity and extensibility of the optimizer [13, 14]. One important theme has been to allow parallel execution in the optimizer [30, 34]. Chaudhuri and colleagues at Microsoft, and Lohman and others at IBM, pioneered the use of the cost estimator component as a tool for recommending better physical structures, eg deciding which indices to create [2, 8–10, 32].

2.5 Evaluating Query Optimization

While much of the literature has taken an engineering approach, designing better ways to do optimization, there has also been work that is scientific in style, looking at how to evaluate the quality of a given optimizer. Our paper is in this tradition. For cost-based optimizers, there has been study of how accurate the estimates are, by comparing the estimated cost to measured cost when a plan is run [16, 21]. This does not of course apply to MongoDB since FPTP doesn't use cost estimates. Another aspect of evaluation is to find a suitable set of queries to test the optimizer on. In this paper we use simple conjunctive queries with two range predicates, but more complex queries are important in practice, and some systems have generated random queries with a mix of predicates and joins, eg [31, 33]. Our work has been most influenced by the work of Haritsa and colleagues, whose focus is on understanding which plan is chosen for each query, especially as selectivity varies. The Picasso tool [19, 28] introduced the plan diagram visulization which is one of the displays we use here. This led to a new goal for an optimizer, to be stable (that is, to have a plan which works well not just for the given query but also for nearby ones) [1].

2.6 Document Data Model

MongoDB is a document-oriented database designed for ease of development and scaling. A MongoDB database stores collections of documents. MongoDB documents (the equivalent of records in a relational DBMS) do not need to have a schema defined beforehand [7, 24]. Instead, the fields in each document can be chosen on the fly. This flexible design allows developers to represent hierarchical relationships, to store arrays, and other more complex structures simply.

Key Components of MongoDB data model design are the database, collection, document and cursor [6]. Table 1 shows the relationship of RDBMS terminology with MongoDB.

Table 1: MongoDB terminology versus RDBMS terminology.

MongoDB	RDBMS
Database	Database
Collection	Table
Document	Record(s)
Field	Column
Embedded Documents	Foreign Key

2.7 MongoDB Query Model

At a low level, MongoDB stores data as BSON documents, which extends the JSON model to offer more data types and efficient encoding and decoding. The programming language APIs make it easy for programmers to create those documents. This property, coupled with the high similarities between MongoDB's JSON-like document model and the data structures used in object-oriented programming, makes integration with applications simple.

MongoDB applications can use complex queries, secondary indexes, and aggregations to retrieve unstructured, semi-structured, and structured data. A vital factor in this flexibility is MongoDB's support for various types of queries. Queries can return documents, projections of documents, or complex aggregations calculated over many documents [24].

Key-value queries The results returned by a key-value query can be based on any field in the document, usually the primary key.

Range queries (i.e. queries with range predicates) The return results returned by a range query are based on values defined as inequality predicates (e.g, greater than, less than or equal to, between).

Text search Text searches are queries which return results in relevance order based on text arguments using Boolean operators such as AND, OR, NOT.

2.8 MongoDB Query Compiler and Optimizer

The design of MongoDB query processor and optimizer is evolving with each release. We provide detailed insights into the state-of-the-art architecture. The overall design of the optimizer is clean and orderly. However, in later sections we demonstrate that there is room for further improvement.

In this section, we explain how a MongoDB query is submitted, parsed and optimized before it interacts with MongoDB storage engine. Explanations in this section are mainly gathered from interviews with professional MongoDB developers and from examining MongoDB source code.

Figure 1 provides an overview of the query processing workflow. We break down the whole process into three layers. In the network layer, MongoDB specifies a MongoDB Wire Protocol which is a simple socket-based, request-response style protocol. Clients connect to the database following the protocol. In the executor layer,

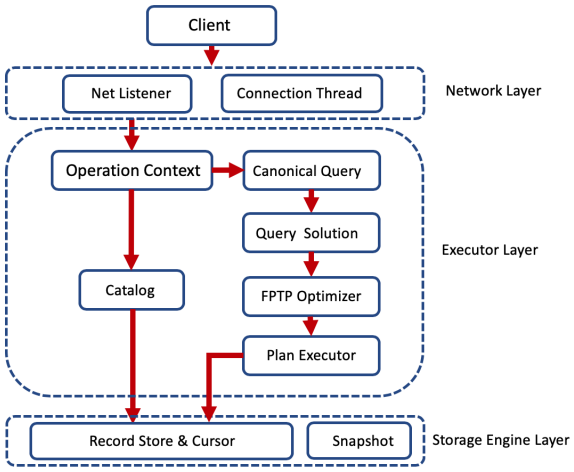


Figure 1: MongoDB query processing workflow.

queries are received in form of operation contexts. Some queries such as `insert()` do not require optimization. Therefore, these queries directly interact with the storage engine.

Queries which require optimization are standardized and simplified to canonical queries, which MongoDB calls the "query shape". For a given Canonical Query, the Query Solution module might yield multiple execution plans. During optimization, if more than one index is associated with the plan, multiple candidate execution plans might be generated. Otherwise, only one execution plan is generated. All candidate execution plans are evaluated by the FPTP query optimizer. Finally, the most efficient candidate plan is executed by the Plan Executor. The plan is also cached, and if the same query shape is executed soon afterwards, the cached plan will be used without considering alternatives.

```
db.runCommand({
    "find": "movie",
    "filter": {
        movieid: {"$gte": 0, "lte":1000},
        avgrating: {"gte": 5}
    },
    "projection": {moviename: 1, avgrating: 1},
    "sort": {avgrating, -1},
    "limit": 10,
})
```

Listing 1: MongoDB find() query example

2.8.1 Canonical Query. Lets have a closer look at MongoDB's `find()` query command and introduce how MongoDB standardizes it. The most common operators for a `find()` command are `projection` (analog to SQL SELECT expression), `filter` (analog to SQL WHERE expression) and `sort`. MongoDB query is different from SQL in many ways - there is no complex semantic parsing, the query is written in BSON format (i.e. Binary JSON, a binary form for representing simple or complex data structures, originated at MongoDB). Each operator in the query is structured such that ordinary queries can be directly taken out.

```
[
    {
        "query" : { "qty" : { "$gt" : 10 } },
        "sort" : { "ord_date" : 1 },
        "projection" : { },
        "queryHash" : "9AAD95BE"
    },
    {
        "query" : { "$or" :
           [
             { "qty":{"$gt":15},"item":"xyz123"},
             { "status" : "A" }
           ]
        },
        "sort" : { },
        "projection" : { },
        "queryHash" : "0A087AD0"
    },
    {
        "query" : { "$or" :
            [
                { "qty" : { "$gt" : 15 } },
                { "status" : "A" }
            ]
        },
        "sort" : { },
        "projection" : { },
        "queryHash" : "DA43B020"
    }
]
```

Listing 2: Examples of query shape

The query is parsed to determine its query shape. The query shape is essentially a query's match expression, projection and sort with the values taken out, as shown in Listing 2. The query shape is designed for the cache mechanism, and it is not used to generate query plans.

The process of generating a canonical query mainly involves match expression optimization. A match expression is an expression which consists of logical operations in the `filter` operator. To optimize a match expression, MongoDB extracts all logical operators and constitutes an expression tree. In this process, MongoDB optimizes the ordering of logical expressions and eliminates duplicates. The simplified expression tree can be used to filter records later.

2.9 MongoDB's FPTP Query Optimizer

We now describe the mechanism of MongoDB's query optimizer. Two vital elements of MongoDB's query optimizer are its FPTP approach and its query plan cache mechanism. A naive explanation of FPTP would be: the query optimizer executes all query plans in parallel and chooses the first one that completes a predefined amount of work. The winner of the race is chosen as the execution plan and cached for future queries with the same shape.

In this work we mainly focus on investigating the query optimizer's query plan evaluation strategy but not the efficiency of its cache mechanism. In the experimental analysis, we pragmatically force MongoDB not to use the query plan cache.

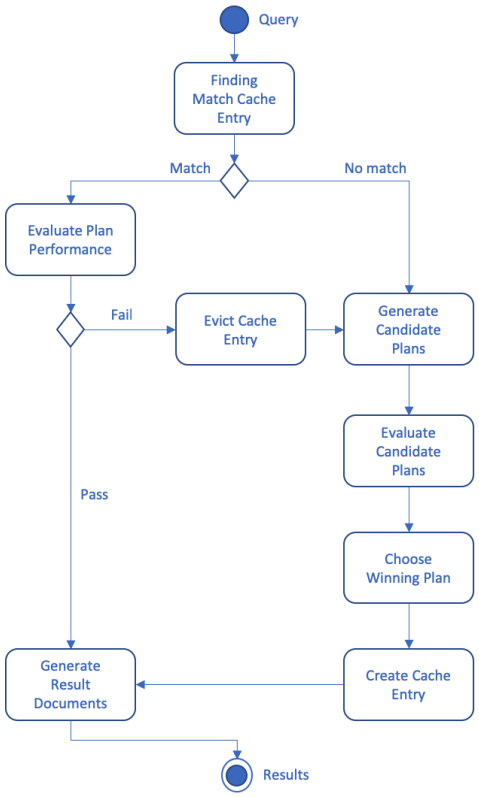


Figure 2: Logical flow of MongoDB query optimizer.

2.10 The First-past-the-post Approach

The FPTP approach consists of two key steps: measuring all candidate plans and then making a decision, i.e. choosing the most efficient query plan based on the measurements. During the benchmarking phase, all possible query plans are executed in RR (round-robin) fashion. RR scheduling is one of the algorithms employed by process and network schedulers in computing. That is to say, MongoDB starts all candidate query plans on the original dataset and cycles through them to simulate the real query plan execution. During the simulation, time slices are assigned to each query plan in equal portions and in circular order, handling all plans without bias. Meanwhile, the query optimizer gathers execution metrics and then provides a score for each plan.

The query optimizer asks each plan for the next document, via a call to a `work()` function. If the plan can supply a document, it responds with `advanced`. Otherwise, the plan responds with `needsTime`. If all documents provided in the simulation have been retrieved, then `isEOF` is assigned with 1. However, the given query could be expensive, so MongoDB specifies limits to terminate the simulation early, as described in Algorithm 1.

The simulation stops if the maximum allowable amount of work has been reached,or the requested number of documents has been retrieved (advanced), or if *isEOF* equals True, i.e. the simulation

Algorithm 1: Algorithm used for early termination of the simulation

```
internalQueryPlanEvaluationWorks ← 10000
internalQueryPlanEvaluationCollFraction ← 0.3
works ← max(internalQueryPlanEvaluationWorks,
internalQueryPlanEvaluationCollFraction ∗
numRecords(collection))
internalQueryPlanEvaluationMaxResults ← 101
needsTime ← 0
advanced ← 0

while i < works do
  if isEOF or
  advanced < internalQueryPlanEvaluationMaxResults then
    break
  end if
  isNextDocRetrieved ← work()
  if isNextDocRetrieved then
    advanced + +
  else
    needsTime + +
  end if
  i + +
end while
```

dataset has no more documents. *internalQueryPlanEvaluationWorks* is the built-in maximum allowable amount of work. However, for very large collections, MongoDB takes a fraction of the number of documents to determine the maximum allowable amount of work. *internalQueryPlanEvaluationMaxResults* is the predefined maximum number of documents that can be retrieved during the simulation.

Metric	Value
baseScore	1
Productivity	queryResults / workUnits
TieBreak	min(1.0 / (10 * workUnits), 1e-4)
noFetchBonus	TieBreak or 0
noSortBonus	TieBreak or 0
noIxisectBonus	TieBreak or 0
tieBreakers	noFetchBonus + noSortBonus + noIxisectBonus
eofBonus	0 or 1

Table 2: MongoDB Query Plan Performance Metrics

When the simulation ends, each candidate query plan is assigned a performance score based on performance metrics shown in Table 2. The query optimizer picks the query plan which has the highest performance score as the execution plan. The performance score is calculated as:

$$score = baseScore + productivity + tieBreakers + eofBonus \quad (1)$$

Each query plan has a base score of 1. The *productivity* is measured as the proportion of the number of results being returned over the total amount of work done. *tieBreakers* is a very small bonus number that is given to no fetch operation (*noFetchBonus*), no blocking sort (*noSortBonus*) and avoiding index intersection (*noIxisectBonus*). *eofBonus* is given if all possible documents are retrieved during plan execution.

3 MEASURING METHODOLOGY

We now describe a detailed methodology to experimentally measure how well a query optimizer chooses query plans. In the remainder of this paper, we will use this methodology to assess the effectiveness if MongoDB's FPTP query optimizer — but note that this approach is not limited to FPTP query optimization as it makes no assumptions about the internal workings of the query optimizer. We also describe the visual presentations we use for presenting the results.

3.1 Testing Setup

The testing environment adopts a typical client-server architecture. A single client communicates with a MongoDB server (build version 4.4.0) which is deployed on an Amazon EC2 (Amazon Elastic Compute Cloud) instance. The instance type we chose is t2.micro. T2 instances work well with Amazon EBS (Amazon Elastic Block Store) gp2 (general purpose SSD) volumes for instance block storage. Gp2 is the default EBS volume type for Amazon EC2 instances and provides the ability to burst to 3,000 IOPS per volume, independent of volume size, to meet the performance needs of most applications and also deliver a consistent baseline of 3 IOPS/GB. 20 GB of storage is sufficient for storing our database and metadata. During the experiments, we need to make sure that RAM is not the bottleneck. For the most stable and fastest processing, we need to ensure that our indexes fit entirely in RAM so that the system can avoid reading the index from disk. Therefore, we configure a RAM size of 1 GB which is more than enough for the data we consider to keep all indexes entirely in-memory.

We wrote a Python client application for executing the experiments and visualizing the results. This evaluation client executes each query and gathers execution statistics from the database using MongoDB's query language. Finally, this client also analyzes statistical information and visualizes the results.

3.2 MongoDB Setup and Database Content

The command line and configuration file interfaces provide MongoDB administrators with a large number of options and settings for controlling the operation of the database system. We modify the default configuration file `mongod.conf` to allow remote access, since our Python application runs on the client side. During the experiment, we use the `cursor.explain()` method with `allPlansExecution` mode to inspect query plan candidates and execution statistics.

The experiments use one collection which contains 1×10^5 documents. Each document has two fields, namely A and B. A positive integer in the range $[0, 1 \times 10^5)$ is assigned to each field. Different experiments use different choices for the contents of the collection, and for the index structures. For example, in the first experiment of Section 4, field A and B both have uniform distribution and the same cardinality with 10^5 distinct values, ie no repetitions among the documents, and we create indices A_1 and B_1 on field A and B, respectively.

3.3 Query Template

In order to assess the effectiveness of the query optimizer, we use conjunctive range queries of the following form:

$$\sigma[lowA \leq A < highA \wedge lowB \leq B < highB](Collection)$$

These are selection queries with two range predicates that allow us detailed control on the query selectivity. All the queries we apply have the same shape, finding all documents whose A field lies within a given range, and also the B field value lies in another range. The ranges are set by parameters *lowA*, *highA*, *lowB*, *highB* whose values determine the selectivities of a particular query of this shape. The queries are then executed using MongoDB's query language as shown in Listing 3.

```
db.collection.find({
    "A" : {"$gte" : lowA, "$lt" : highA},
    "B" : {"$gte" : lowB, "$lt" : highB}
})
```

Listing 3: Query Shape in MongoDB Query Syntax

We can determine the selectivity for range predicate on field X, e_X, using the following formula:

$$e_X = \frac{numbers\ of\ documents\ satisfying\ range\ predicate\ on\ field\ X}{total\ number\ of\ documents} \tag{2}$$

3.4 Methodology

For each experiment, we create a single client and one Amazon EC2 instance with a MongoDB database installed. We then build a database with the appropriate data distribution, and create the indices for the particular experiment.

Then, the client performs queries which have different selectivity on A and B. More specifically, each query initiated by the client is a conjunction of one range predicate on A and the other one on B.

For each query, the client records the selectivity for the range predicates on field A & B, the execution plan, and the time consumption of seeking and executing the plan.

The client generates random queries of the given shape, by picking random values for the parameters, to circumvent any potential workload prediction strategies by the query optimizer. For each query, the client determines and tracks the selectivity of the predicate on field A, and the selectivity of the predicate on field B. It then submits the query to MongoDB and the optimizer will determine the query plan using FPTP and then execute this plan. We have disabled query plan caching for these experiments, so that MongoDB does the optimization properly for every submitted query. The client determines which plan MongoDB chose for the query, using the MongoDB `explain` method.

We also determine the runtime cost of each reasonable query plan, so that we can determine which is really the best (and thus we will see whether or not MongoDB is choosing wisely). MongoDB provides users with a `hint(<queryPlan>)` method to customize execution plan selection. We use this method to force the query optimizer to execute each suitable plan. We run each plan ten times; when doing this, we extract the `executionStats.executionTimeMillis` field to retrieve the wall-clock time in milliseconds required for each query execution. We then average the execution times for a plan, and we identify the optimal query plan: it is the one that has lowest average execution time, among the possible plans for this query.

Outlier handling. When we gather execution times of all query plans, we note that there are a few outliers which deviate markedly from other data points. These outliers are usually fifty times higher than the mean value. We suppose that the likely cause is unstable performance by our T2.micro instances rather than inherit in the benchmarked database. Therefore, we identify those outliers, and drop them using the standard 1.5 IQR (interquartile range) rule. We measure the interquartile range for $t'_{i,j,1}, t'_{i,j,2}, ..., t'_{i,j,N}$ and then multiply the interquartile range (IQR) by the factor 1.5. Any data point greater than the sum of the third quartile and 1.5 IQR is dropped; any number less than the difference of the first quartile and 1.5 IQR is dropped as well.

Accuracy and Impact Metrics. We finally assess the effectiveness of a query optimizer by two metrics: The measure of the overall success of MongoDB's optimizer is its *accuracy*: the fraction of queries for which the optimizer's chosen plan is actually the best one.

As well as finding whether or not MongoDB's FPTP optimizer chooses the best plan, we also want to know how much query performance is impacted by the choice. If the chosen plan were only slightly slower than the best possible plan, users might find this quite acceptable, but if MongoDB chooses a plan that is much slower than another, this is more serious. So, for each query, we determine the ratio of the runtime of the plan chosen by the optimizer, to the runtime of the plan which actually runs fastest for that query. We can consider this for each query, and we also calculate the average over all the queries we use, as the *impact* of the optimizer.

3.5 Visualization of Plan Choices

We visualize the outcomes of our query optimizer evaluation using two kinds of visual plots. The first visual display technique is inspired by the "plan diagrams" used by the Picasso system [28]. The central idea is to consider a family of conjunctive queries with two range predicates which are parameterised by the selectivity of two attributes. For each choice of selectivities, we identify which query plan is chosen by the optimizer, and we plot that decision on a square diagram, where the x and y coordinates reflect the selectivities in that query. Each potential plan is indicated by a different color that can be assigned to the point. We illustrate the plan chosen by MongoDB by the color displayed in a cell of an appropriately discretized square grid. We use the same style of diagram for the plan actually chosen by the optimizer, and also for illustrating which plan would truly be the fastest for the query. This way, readers can easily visually compare whether the diagrams are similar to one other (indicating that MongoDB usually chooses the truly best plan) or very different.

We describe how a choice of plan is displayed in a plan diagram, following [28]. For each query the client will record e_A, e_B and P_{e_A,e_B} (i.e. the execution plan for the query with selectivity e_A and e_B). The client will map P_{e_A,e_B} to a position (i, j) in the grid. The magnitude of i and j is proportional to e_A and e_B. Different execution plans are represented by pixels with different colours, as

demonstrated in Figure 3 (which shows the optimal choice of plan from our first set of experiments). In this example, there are only three possible execution plans, so we use three contrasting colours to distinguish them:

IXSCAN_A an index scan uses index A_1 on field A. This execution plan is represented by an orange pixel.

IXSCAN_B an index scan uses index B_1 on field B. This execution plan is represented by a green pixel.

COLLSCAN a collection scan (i.e. a table scan). This execution plan is represented by a yellow pixel.

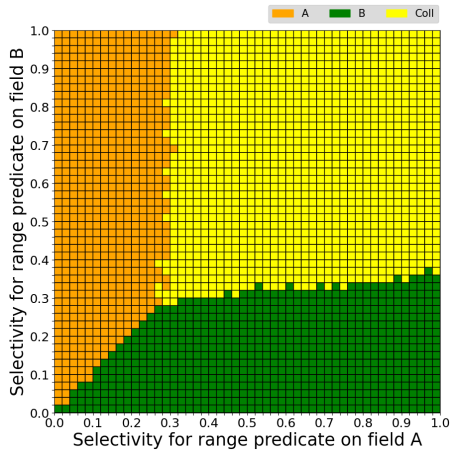


Figure 3: Visual mapping of execution plans for different selectivities.

The grid has dimension 50×50 and initiated with grey so that we can identify the abnormal behaviour of the query optimizer (i.e. timeout or exceptions are thrown). When the experiment starts, the client keeps generating queries and trying to fill the grid with three colours, until all 50^2 positions in the grid have been visited. Algorithm 2 describes how we map a pair of selectivity to a position in the grid. Algorithm 3 explains the entire visual mapping process.

Algorithm 2: Mapping a pair of selectivities to coordinate

e_A, e_B {selectivity for range predicates on A and B}
$i, j \in \mathbb{Z}^+$ {column and row number in the grid}
$D \leftarrow 50$ {dimension of the grid.}
$\delta_e \leftarrow D * \frac{100}{D}$ {step of our measurement}
$i \leftarrow e_A / \delta_e$
$j \leftarrow e_B / \delta_e$

3.6 Visualization of Performance Impact

We further want to assess how much the query performance is impacted by the choices the optimizer makes. To do so, we determine for each query the ratio of the runtime of the plan chosen by the optimizer to the runtime of the plan which actually runs fastest for that query. We propose a heatmap presentation (which we believe has not been used before) for this aspect of optimizer effectiveness, in which we plot these ratios on the same discretized grid, with

Algorithm 3: Algorithm used for visual mapping

$D \leftarrow$ Dimension of the grid
$M_p \leftarrow$ 2D array of -1 {with dimension D, execution plans expressed by different color}
$M_t \leftarrow$ 2D array of -1 {with dimension D, time of seeking execution plan}
$M_{visited} \leftarrow$ 2D array of 0 {with dimension D, locations has been visited.}
$v \leftarrow 0$ {Number of locations have been visited}
$N \leftarrow D^2$ {total number of locations}
$a_{min}, a_{max} \leftarrow$ min, max value of field A
$b_{min}, b_{max} \leftarrow$ min, max value of field B
e_A, e_B {selectivity for range predicate on A and B}
t_p {time of seeking execution plan}
P {execution plan}
Q {MongoDB query}

while $v < N$ **do**
  $a_{lower}, a_{upper} \leftarrow$ generateRandomRangePredicate(a_{min}, a_{max})
  $b_{lower}, b_{upper} \leftarrow$ generateRandomRangePredicate(b_{min}, b_{max})
  $e_A \leftarrow$ computeSelectivity(a_{lower}, a_{upper})
  $e_B \leftarrow$ computeSelectivity(b_{lower}, b_{upper})
  $i, j \leftarrow$ mapSelectivityToCoordinate(e_A, e_B)

  if $M_{visited}[j][i] == 1$ **then**
    continue
  end if

  $Q \leftarrow$ generateQuery($a_{lower}, a_{upper}, b_{lower}, b_{upper}$)
  $P, t_p \leftarrow$ db.collection.find(Q).explain()

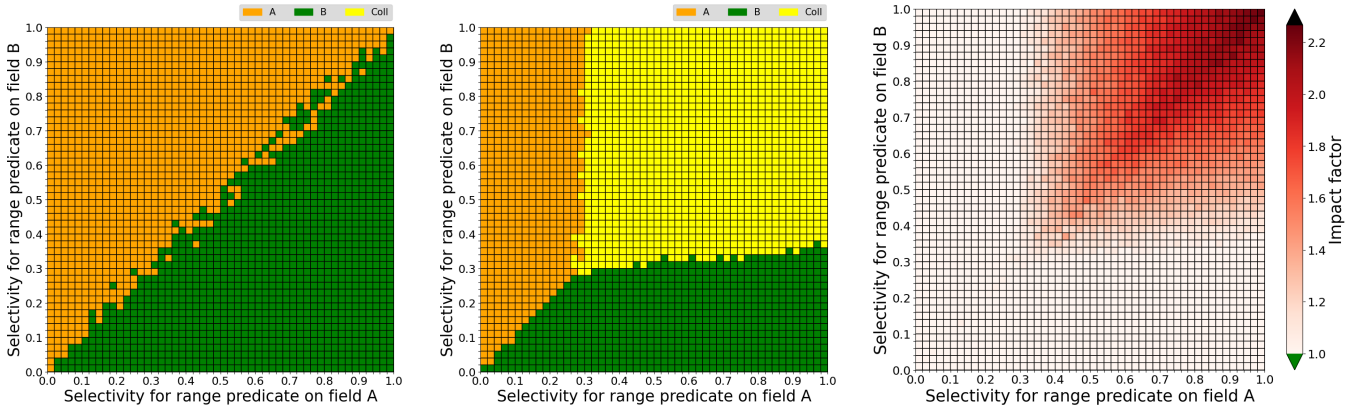
  $M_p[j][i] \leftarrow$ colorOfPlan(P)

  $M_t[j][i] \leftarrow t_p$
  $M_{visited}[j][i] \leftarrow 1$
  v ++
end while

coordinates reflecting the selectivities, and we use a heatmap for the color, in which the color saturation corresponds to the ratio between runtime of the chosen plan and runtime of the truly optimal plan for that query. If the query optimizer has chosen the optimal plan (a performance ratio of 1.0 between its choice and the optimal plan), we represent this with white pixels. But the darker the colour of a box, the worse the performance impact of the query optimizer's (sub-optimal) choice. An example for such a heatmap visualizing the FPTP's performance impact is shown in Figure 4(c).

4 EVALUATION OF MONGODB'S FPTP QUERY OPTIMIZER

In the following, we use our visualization approach from Section 3 to assess the effectiveness of FPTP query optimization in MongoDB, in which we see which plans are chosen and which plans would



(a) Chosen query plan with both attributes indexed: MongoDB never chooses a collection scan.

(b) Optimal query plan for different selectivities with both attributes indexed.

(c) Performance impact of MongoDB's sub-optimal choices.

Figure 4: Effectiveness of MongoDB's query optimizer with conjunctive filter queries and both attributes indexed.

actually be the best. We use varying physical designs for the data (varying sets of indexes).

4.1 Querying Indexed Collections

In our first experiment, we investigate the case of conjunctive filter queries over two attributes where both of the queried attributes are indexed. The physical data design is, as described in Section 3, a collection with two integer-valued fields (A and B). In this experiment, each field has uniform data distribution, and each field has an index. Therefore there are three reasonable query execution plans. One is to retrieve the documents that match the predicate on A, using the index on A, and then examine each to see whether the document also meets the predicate on B; this plan is called IXSCAN_A, an index scan on A. We can symmetrically do an index scan on B using the index on B (IXSCAN_B). We can instead perform a full collection scan, examining each document to see whether it meets the conjunction of both predicates (COLLSCAN).

Figure 4(a) plots the execution plans picked in this case by the query optimizer of MongoDB 4.4.0. We observe that both IXSCAN_A and IXSCAN_B are picked by the optimizer with equal chance, and as expected, the field where the query selectivity is lower (that is, fewer documents satisfy this predicate), has the index used. This phenomenon suggests that the FPTP approach is capable of making efficient choices between index scans. The boundary between the two index scans is clear, and it forms a diagonal across the grid, which demonstrates that the query optimizer has a quite robust performance (the chosen plan doesn't usually change with small perturbations in the query).

However, we see that during all the runs of this experiment, the COLLSCAN was never the chosen plan. To see that this is surprising, we present the diagram showing which plan is actually the best for the queries in this physical design (in Figure 4(b), which is the same as Figure 3) and what the ratio is between the cost of running the chosen plan compared to the true best plan (Figure 4(c)).

As one would expect, the performance of an index scan is significantly worse than a collection scan for a query that has very high selectivity on both fields (that is, each predicate is satisfied by most of the documents). At the extreme, if the query retrieves all documents in the collection, an index scan would firstly retrieve index documents, and then use those index information to locate and fetch all documents. In contrast, a collection scan does not have the index-access overhead; it directly retrieves all documents without using any index. Therefore, the top-right corner of the grid in Figure 4(b) is yellow as expected. Since MongoDB does not choose the collection scan, in this corner of Figure 4(c), we see strong red, indicating the chosen query runs substantially slower than the best possible execution plan.

In summary, MongoDB's FPTP query optimizer shows an overall accuracy of just shy of 52% in this scenario, so about half the time there would be a better plan than one of the chosen index scan plans. The average impact factor of all those optimizer decisions is 1.24 (in average, the chosen query plan is 24% slower than the optimal plan), with some plans more than twice as slow compared to a collection scan.

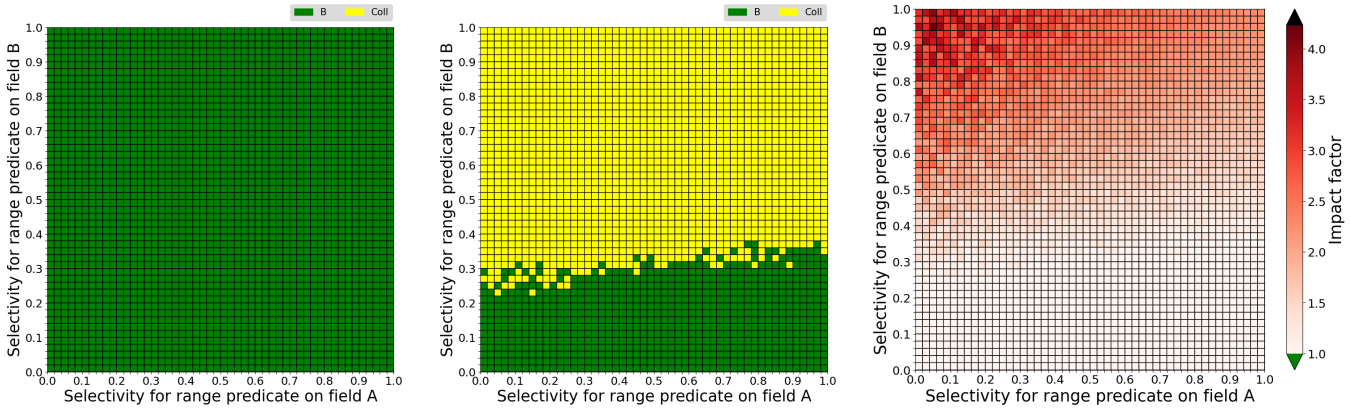
4.2 Single Index Scenario

In the next experiment, we investigate the behavior of MongoDB's query optimizer with a physical structure where there is an index for only one of the attributes (field B). Again we have uniformly distributed values in each field.

Figure 5(a) and Figure 5(b) plot the query plans chosen by MongoDB 4.4.0 and the optimal query plans, respectively. Figure 5(c) shows the performance impact of the preference bias issue.

From Figures 5(a) and 5(b) we see that the collection scan is actually best as long as more than about 30% of records satisfy the range predicate on B; however even in these situations MongoDB chooses to use the available index on B; again it never does a collection scan. Thus the optimizer's choice is even less accurate than in the previous scenario, with an overall accuracy of just about 30%.

And these sub-optimal choices sometimes come with a huge cost. Near the boundary between optimal choices, the two have similar costs (and so there is not much impact from choosing the index scan), however, in cases where most documents satisfy the predicate on B, we find that the plan chosen by MongoDB (the

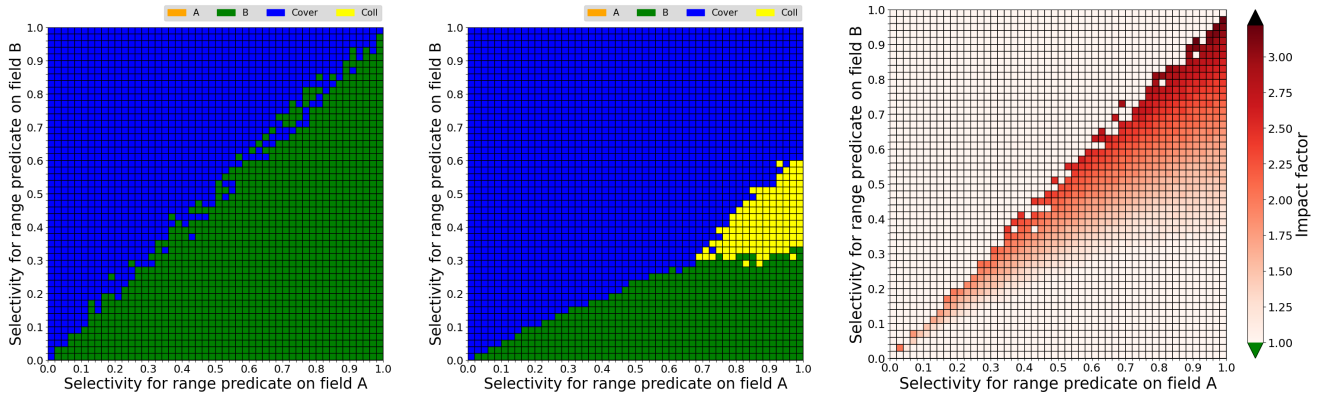


(a) Chosen query plan with only attribute B indexed: MongoDB still never chooses a collection scan.

(b) Optimal query plans for different selectivities with only one index.

(c) Performance impact of MongoDB's query plan choices.

Figure 5: Effectiveness of MongoDB's query optimizer with conjunctive filter queries and only one attribute indexed.



(a) Chosen query plan with all attributes indexed and covering index available too.

(b) Optimal query plans for different selectivities with covering index available too.

(c) Performance impact of MongoDB's sub-optimal plan choices.

Figure 6: Effectiveness of MongoDB's query optimizer with all queried attributes being indexed including a covering index.

index scan) can execute up to 343% slower than the best plan for that query. The average slowdown of MongoDB's choice compared to the best possible, is 68%.

4.3 Covering Index Scenario

In our third experiment, we run our usual conjunctive queries over two attributes against a physical database design where, as well as an index on each attribute, there is also a covering index on the combination (A,B) in that order. The power of the covering index is that query execution often need not access the document at all, as the information in the index is enough to determine whether or not the document matches both ranges in the predicate.

Figure 6 shows the results of this third experiment. We again first visualize the query plans chosen by MongoDB's FPTP optimizer (Figure 6(a)), the middle plot shows the optimal plan (Figure 6(b)), and the third plot is a visualization of the performance impact of MongoDB's sub-optimal plan choices (Figure 6(c)).

From the colour distribution in Figure 6(b), we can see that an execution plan using the covering index would indeed be fastest in most cases, and that the plan space where a full collection scan is best is now notably smaller than in the previous two scenarios. Furthermore, Figure 6(a) shows that MongoDB's FPTP optimizer indeed favours a covering index over a plan using an index just on A. But again we see that FPTP does not choose the collection scan even in cases where it is best, and also, that it often chooses a plan that access through the index on B rather than the plan using the covering index. We see substantial inaccuracy, and many cases where the performance is markedly worse that what the true optimal plan would deliver. While the overall accuracy of FPTP is the best of all three scenarios with about 71%, the average query speed is still slower than it could be, by 23%.

4.4 Performance with Varying Data Distributions

Our experiments so far always queried two numerical attributes with a uniform data distribution. We also did some extensive experiments with varying the data distributions of each attribute; for

example each field A and B could have uniform, normal or zipfian distributions. However, we did not find any different behaviour of MongoDB's query optimizer in these experiments than shown so far for a uniform data distribution. The reason probably is that MongoDB chooses its 'best' plan as soon as the first candidate plan produces the 100th result row, which is too early to show any significant difference between the varying data distributions. For space reasons, we omit the details of these runs from this paper and refer the interested reader to our forthcoming technical report.

Evaluation Summary. Overall, the experiments in this section demonstrate that while FPTP is a viable approach to query optimization with many good plan choices, it also suffers from a clear bias towards index scans over full collections scans. The question is whether this is simply an implementation bug in the specific version of MongoDB (v4.4.0), or whether it points towards an inherit weakness of FPTP query optimization. We hence take a closer look into MongoDB's query optimizer in the next section.

5 WHY DOES MONGODB'S FPTP OPTIMIZER AVOID COLLECTION SCAN?

As we just saw in Section 4, MongoDB's FPTP optimizer doesn't choose collection scan plans, even for queries when it would run substantially faster than using an index. In this section, we take a closer look at the code of the optimizer to identify reasons for this preference bias issue in MongoDB.

A closer inspection of the query optimizer code revealed a surprising design choice:

by default, MongoDB 4.4.0 does not include collection scan among the list of candidate plans to be run in FPTP, if an index is available to satisfy a query. In more detail, line 1082 of `src/mongo/db/query/query_planner.cpp` [1] is the check before a collection scan is included:

```
if (possibleToCollscan &&
   (collscanRequested || collScanRequired)) { ...
```

The variable `possibleToCollscan` indicates whether a collection scan is possible (database administrators can disable collection scans or the query can include a hint that requires use of an index); `collscanRequested` indicates that an explicit query hint has specified that a collection scan should be used; and `collScanRequired` is true only if there is no matching index. In other words, if a matching index exists, a collection scan must be explicitly requested for MongoDB to consider it. If the collection scan isn't tasking part, it can't win the race.

5.1 Forcing consideration of the collection scan

However, is this all there is to the issue? We modified the source code to produce a variant we call MongoDB+COLLSCAN that simply always adds a COLLSCAN plan to the set of candidate plans which MongoDB's FPTP optimizer tries out. We repeated our experiments with this variant dbms, and we did not see substantial differences in the chosen plans, from unmodified MongoDB 4.4.0 (cf. Figure 7). That is, even when COLLSCAN is considered in the FPTP race, it does not win, even for queries where the collection scan plan runs substantially faster than an alternative plan with index scan.

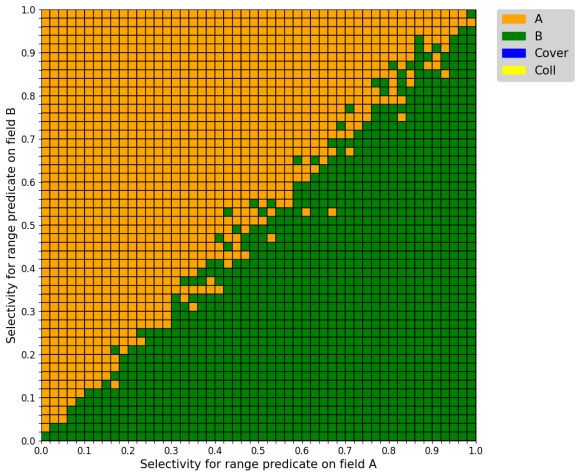


Figure 7: Chosen plans by MongoDB+COLLSCAN.

5.2 Overrated Index Scan

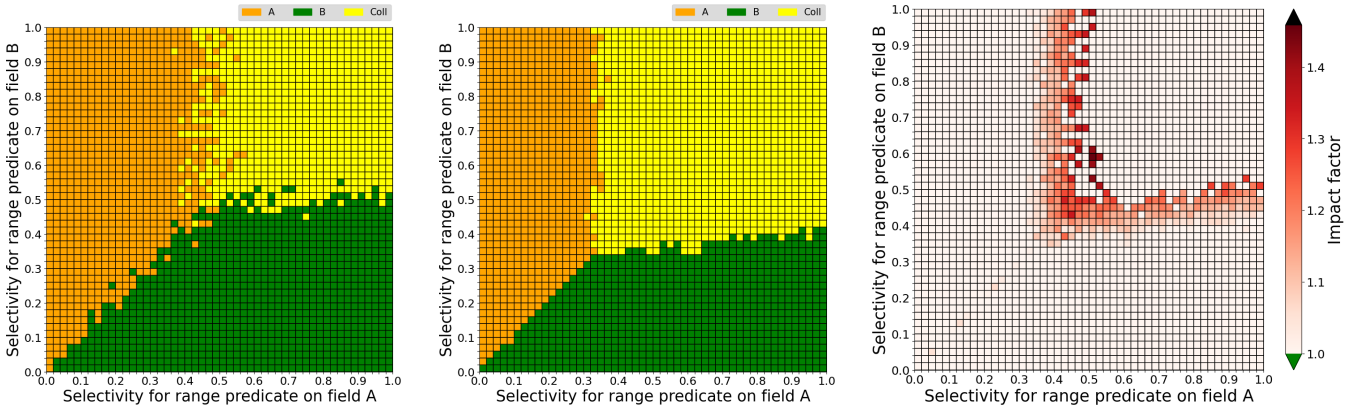
To explore further the cause of the preference bias that over-rates the index scan compared to collection scan, we looked in detail at the query execution log showing the activity of MongoDB+COLLSCAN during the optimization race for a query with selectivity 0.95 in each attribute. In the visualizations above, we see that for this query collection scan is clearly superior, but the optimizer chooses to use an index. As we mentioned in Section 2, the FPTP approach assigns a score to summarize the performance of each query plan at the end of the race (and then the query plan with the highest score will be chosen). The formula considers the *productivity* of each query plan, where productivity is based on the ratio between result documents produced and the work units performed during the race. Note that these work units represent logical costs, not actual measured runtimes.

As it turns out, when determining the workUnit of an index scan, the MongoDB implementation of the FPTP approach ignores the cost of fetching index documents; the optimizer in MongoDB 4.4.0 treats the index retrieving work and the document retrieving work together as a single unit of work (i.e. the same amount of work required by a collection scan looking at the same number of documents). Therefore, the workunit is under-counted for the index scan and so the productivity of an index scan is overrated. This implementation detail in MongoDB's query optimizer code hides the true advantage of a collection scan in many cases. We suggest that it would be appropriate for MongoDB to adjust the way workunit is measured in the race, to correct this.

5.3 Adjusting productivity score of index scan

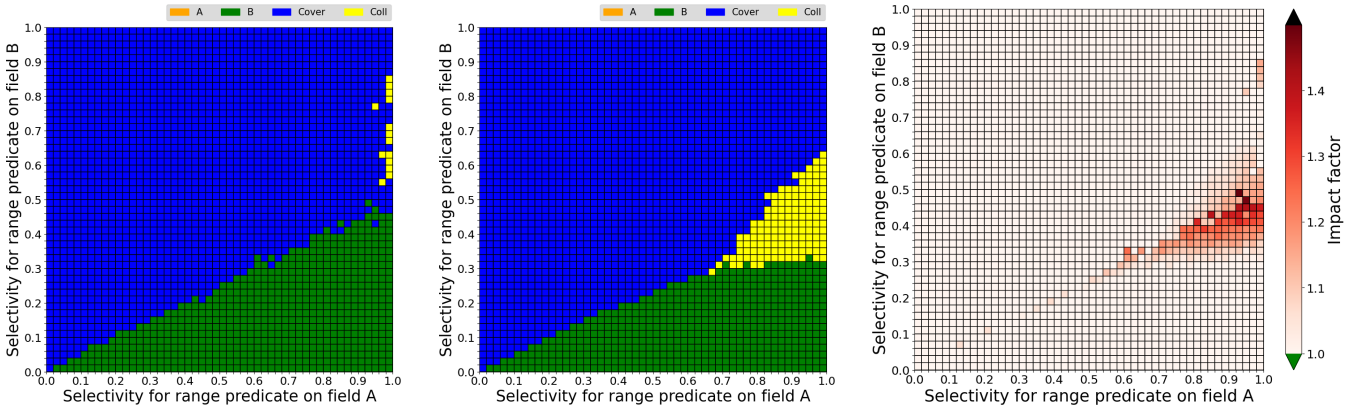
We showed above that the optimizer ascribes too much productivity to the index scan, because it treats as 1 workunit, the combination of retrieving the index entry, and retrieving the document the index points to. So we made another small modification to the optimizer code. When the plan contains a FETCH (that is, when there is

[1] https://github.com/mongodb/mongo/blob/r4.4.0/src/mongo/db/query/query_planner.cpp#L1082



(a) Plans chosen by MongoDB_MOD. (b) Optimal Plan Choices. (c) Performance Impact of MongoDB_MOD's choices.

Figure 8: Effectiveness of modified FPTP query pptimizer of MongoDB_MOD (dual index scenario).



(a) Plans chosen by MongoDB_MOD. (b) Optimal Plan Choices. (c) Performance Impact of MongoDB_MOD's choices.

Figure 9: Effectiveness of modified FPTP query optimizer of MongoDB_MOD (covering index scenario).

a lookup through the index) we simply halved the productivity score calculated. This is overly simplistic for complex query plans, but its a reasonable shortcut for the simple queries in these experiments. We call the variant system with forced consideration of COLLSCAN, and also with the adjusted productivity score as described, MongoDB_MOD. This system is evaluated for the physical design with uniform data distribution and two indices, in Figures 8(a)–8(c). We see that, while the modified score is not a perfect adjustment, it gets the right decision in many of the queries, and the chosen plan is never worse by much than the best possible. We remark that, in theory, the optimal plan choice ought to be exactly the same for this system as for unmodified MongoDB 4.4.0 since the only change is in the optimiser rather than in query plan execution (that is, Fig 8(b) should be the same as Fig 4(b). This is not exactly the case, because each figure is produced from measurements of the running time of the plans, and there is some experimental variation from run to run; however the differences are only occasional cells near the region boundary, where two plans have almost the same cost, so which is fastest (by a tiny amount) can vary between runs.

The overall accuracy of the query optimizer of this modified version of MongoDB is now 85% (up from just 52% measured in Section 4.1), with an average performance impact of the remaining sub-optimal plan choices of just 2%.

The suggested adjustment of the score of index scans also helps in the scenario with a physical design including covering indexes, as shown in Figure 9. If we compare the plan choices of the modified FPTP optimizer in Figure 9(a) with the ones done by the original MongoDB 4.4.0's optimizer in Figure 6(a), we not only see collection scans sometimes being chosen, but also note the much broader use of the covering index which resembles much closer the optimal case (the few variations near the region boundaries between the optimal cases between Figures 9(b) and 6(b) are due to slight variations in the runtimes between experiments). Consequently, the remaining sub-optimal plan choices by FPTP have a much reduced performance impact (cf. Figure 9(c)). The overall accuracy of the modified query optimizer is now 91%, with an average performance impact of just 1%.

5.4 Discussion

We have shown that the coding in MongoDB's FPTP query optimization has a systematic preference bias that can lead to poor choice of execution plan. The outcomes can be improved by forcing consideration of COLLSCAN in the race, and adjusting the productivity score to recognize the extra work done when both index lookup and then document lookup happen. Further work is needed to find more sophisticated ways to score productivity, that will deal with complicated plans with index lookup on some but not all steps.

We know that the scale of our workload is considerably smaller than common in real-world settings; we only measured cases where the indices all fit in memory. In reality, companies often have Terabytes of data stored in MongoDB, and they have more substantial index structures. This would significantly magnify the true costs of the index scan and thus dramatically increase the negative impact of the preference bias from neglecting the cost of fetching index entries. Therefore, in such cases, we expect that the preference bias becomes a much more serious issue.

We ran earlier experiments on MongoDB 4.0.12, and we found another mistake in the code that ran the race. The count of records retrieved for the collection scan was initialised at -1 rather than as 0. This led to the collection scan always retrieving one fewer record when the race concluded. This bug has now been fixed in MongoDB 4.4.0 which we measure in this paper.

6 CONCLUSIONS

MongoDB uses a unique approach to query optimization, which we term FPTP query optimization. This differs significantly from the traditional cost-based query optimization in that it chooses its execution plans based on an "execution race" with multiple candidate plans. To the best of our knowledge, this is the first paper explaining and evaluating this approach.

We analysed the effectiveness of MongoDB's FPTP optimizer using experiments that consider a set of queries, which adjust parameter values to vary the selectivity. For each query, we find which plan the MongoDB optimizer chooses, which plan is actually the fastest in execution, and we see how much worse the chosen plan is compared to the optimal one. Each of these aspects is displayed on a grid of cells. These displays provide two plan diagrams [28] and an innovative heatmap visualization of the impact on performance of the optimizer's choices. This visualization makes it easy to visually identify areas where query planning can be improved.

We demonstrated that MongoDB 4.4.0 has a query plan preference bias: the current implementation of FPTP does not even consider a collection scan among the a candidate plans, unless it is requested to do so by the client, or there is no alternative. Furthermore, we showed that MongoDB calculation of plan productivity (during the race) underestimates the costs of an index scan where both index entry and document must be accessed. This leads MongoDB to choose an index-based plan that can be substantially slower than an optimal collection scan, for some queries that retrieve many of the documents.

This paper offers the beginning of a deep study of the MongoDB query optimizer. So far we have only considered simple queries that run over a collection of documents with two atomic fields, and use range predicates on each field. In future work we will examine how the optimizer works on more complicated document schema, and more complicated queries. Our suggested adjustment of the productivity score to account for index access as well as document access, will also need to become more nuanced, before it would be appropriate for production use on a wider class of queries. Our approach could form the basis of an automated regression testing tool to verify that the query planner in MongoDB improves over time.

REFERENCES

[1] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. 2010. On the Stability of Plan Costs and the Costs of Plan Stability. *Proc. VLDB Endow.* 3, 1 (2010), 1137–1148.

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 930–932.

[3] Rafiul Ahad, KV Bapa, and Dennis McLeod. 1989. On estimating the cardinality of the projection of a database relation. *ACM Transactions on Database Systems (TODS)* 14, 1 (1989), 28–40.

[4] Brian Babcock and Surajit Chaudhuri. 2005. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 119–130.

[5] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM, 107–118.

[6] Kyle Banker. 2011. *MongoDB in action*. Manning Publications Co.

[7] Deka Ganesh Chandra. 2015. BASE analysis of NoSQL database. *Future Generation Computer Systems* 52 (2015), 13–21.

[8] S. Chaudhuri, M. Datar, and V. Narasayya. 2004. Index selection for databases: a hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering* 16 (2004), 1313–1323. https://doi.org/10.1109/tkde.2004.75

[9] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. *ACM SIGMOD Record* 27 (1998), 367–378. https://doi.org/10.1145/276305.276337

[10] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. 2008. A pay-as-you-go framework for query execution feedback. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1141–1152.

[11] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.

[12] Johann Christoph Freytag. 1987. A Rule-Based View of Query Optimization. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. 173–180.

[13] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.

[14] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[15] Goetz Graefe and David J. DeWitt. 1987. The EXODUS Optimizer Generator. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*. 160–172.

[16] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2012. Testing the accuracy of query optimizers. In *Proceedings of the Fifth International Workshop on Testing Database Systems, DBTest 2012, Scottsdale, AZ, USA, May 21, 2012*. 11.

[17] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. [n.d.].

[18] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. 1996. Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.* 52, 3 (1996), 550–569.

[19] Jayant R. Haritsa. 2010. The Picasso Database Query Optimizer Visualizer. *Proc. VLDB Endow.* 3, 2 (2010), 1517–1520.

[20] Tapio Lahdenmaki and Mike Leach. 2005. *Relational Database Index Design and the Optimizers: DB2, Oracle, SQL Server, et al.* John Wiley & Sons.

[21] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[22] Volker Markl, Nimrod Megiddo, Marcel Kutsch, Tam Minh Tran, Peter J. Haas, and Utkarsh Srivastava. 2005. Consistently Estimating the Selectivity of Conjuncts of Predicates. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*. 373–384.

[23] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. 2004. Robust query processing through progressive

optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 659–670.

[24] MongoDB. 2019. MongoDB Architecture Guide: Overview. https://www.mongodb.com/collateral/mongodb-architecture-guide

[25] Frank Olken and Doron Rotem. 1995. Random sampling from databases: a survey. *Statistics and Computing* 5, 1 (1995), 25–42.

[26] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*. 39–48.

[27] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*. 294–305.

[28] Naveen Reddy and Jayant R Haritsa. 2005. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 1228–1239.

[29] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*. 23–34.

[30] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. 337–348.

[31] Michael Stillger and Johann Christoph Freytag. 1995. Testing the Quality of a Query Optimizer. *IEEE Data Eng. Bull.* 18, 3 (1995), 41–48.

[32] G. Valentin, M. Zuliani, D.C. Zilio, G. Lohman, and A. Skelley. 2000. DB2 advisor: an optimizer smart enough to recommend its own indexes. *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)* (2000), 101–110. https://doi.org/10.1109/icde.2000.839397

[33] Florian Waas and César A. Galindo-Legaria. 2000. Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. 499–509.

[34] Florian M. Waas and Joseph M. Hellerstein. 2009. Parallelizing extensible query optimizers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. 871–878.