

Article

MongoDB Aggregation Pipeline Performance: Analysis of Query Plan Selection and Optimizer Behavior Across Versions and Collection Scales

Rosen Ivanov 

Department of Computer Systems and Technologies, Technical University Gabrovo, 5300 Gabrovo, Bulgaria; rosen@tugab.bg

Abstract

This article examines how MongoDB optimizes aggregation pipeline queries, focusing on two mechanisms: a trial-based plan selection process that runs candidate execution plans in parallel and picks the one returning the most results for the least work, and rule-based operator rewriting by the Pipeline Optimizer. The study tests nine aggregation query types on a synthetic e-commerce dataset with 50K documents, using MongoDB versions 6.0.3 and 8.2.5 under identical conditions. For each query, all valid operator orderings are evaluated together with the physical execution plan and the Pipeline Optimizer output. Each test runs 20 times with the plan cache cleared before every run. The study also tests scalability with datasets of 150K and 250K documents. Three cases are identified where the rule-based optimizer falls short: IXSCAN preference bias at low selectivity, where the suboptimal plan is up to nine times slower than the optimal (80 ms vs. 699 ms at 250K under MongoDB 8.2.5), unbounded document multiplication after \$unwind, and failure to account for \$group output cardinality. MongoDB 8.2.5 improves performance in most cases compared to version 6.0.3. \$match + \$group queries run up to 28% faster. Queries that rely on IXSCAN improve by up to 18%. Unbounded projection operations run slower in MongoDB 8.2.5 at all tested sizes. The slowdown is +23% at 50K, +3% at 150K, and +14% at 250K, pointing to a change in the projection execution path between versions.

Keywords: MongoDB; aggregation pipeline; query optimization; plan selection; Pipeline Optimizer; index selection; scalability



Academic Editors:
Christoforos Kachris and Heming Jia

Received: 28 March 2026

Revised: 30 April 2026

Accepted: 12 May 2026

Published: 15 May 2026

Copyright: © 2026 by the author.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

The amount of unstructured data produced by modern web applications keeps growing, and document-oriented NoSQL databases have become a practical answer to this problem. MongoDB—which uses BSON format and requires no fixed schema—has established itself as one of the leading systems in this category, running thousands of production applications across different industries [1]. As more organizations move their analytical work to MongoDB, the speed of query execution becomes increasingly critical. Aggregation pipelines with multiple stages present a particular challenge: depending on how operators are ordered and which execution plan is chosen, the same query can run in milliseconds or take seconds.

Query optimization has been a longstanding challenge in database research for a long time [2]. Relational databases like PostgreSQL, Oracle, and SQL Server have historically tackled it with cost-based optimization (CBO), a method that goes back to the System R

project. The approach works as follows: generate candidate execution plans, estimate the cost of each using statistics like column histograms and table sizes, and then run the one with the lowest estimated cost. The problem is that this relies on accurate cardinality estimates, and in practice, they are often wrong. Leis et al. [3] tested this directly using the Join Order Benchmark on the IMDB dataset and found that even mature, production-grade optimizers regularly produce cardinality errors of several orders of magnitude. The main culprit is a pair of simplifying assumptions: that predicates are independent, and that data is uniformly distributed. Their conclusion was clear—bad cardinality estimates do more damage to query performance than an imprecise cost model, and this finding has driven much of the subsequent work on learned query optimization.

MongoDB takes a fundamentally different approach. Instead of estimating costs before execution, its optimizer runs the candidate plans directly—a strategy that Tao et al. [4] call First Past the Post (FPTP). When a query arrives, MongoDB generates multiple candidate plans based on available indexes and query structure, then executes them all in parallel for a short trial period. Each plan receives the same work budget, counted in iteration units under the document-iterator model of the classic execution engine. The plan that returns the most documents relative to the work it performs is selected, executed to completion, and stored in the plan cache for future queries of the same shape. The key advantage of this approach is that it needs no statistics and measures actual runtime behavior—which makes it resilient to data skew. The limitation is that the trial period covers only a small fraction of full execution, which can introduce systematic biases in plan selection.

MongoDB supports two ways to query data. The simpler one is `find()`, which filters and returns documents but cannot perform computations across multiple documents. The aggregation pipeline goes well beyond this: it is a sequence of processing stages, each transforming the document stream produced by the previous one, and it supports operations such as filtering (`$match`), sorting (`$sort`), limiting the result count (`$limit`), projecting selected fields (`$project`), grouping documents and computing aggregates (`$group`), unnesting array fields into separate documents (`$unwind`), and joining with other collections (`$lookup`). The most used aggregation operators and their purposes are described in Supplementary Materials/Table S1. Internally, the pipeline splits into two layers that have fundamentally different relationships to the query optimizer. The first is the cursor stage, which contains the physical operators that run directly against the storage engine and can take advantage of indexes—operators like `$match`, `$sort`, and `$limit` fall into this layer and participate in the FPTP plan selection race described above. The second layer contains the remaining pipeline stages—`$group`, `$unwind`, `$lookup`, and any `$project` that cannot be satisfied through an index—which execute after the cursor stage on the document stream it produces and sit entirely outside the FPTP mechanism. This split is consequential: only the operators in the cursor layer influence which execution plan is selected, while the operators in the second layer are invisible to the plan selection process regardless of how much they affect the result.

MongoDB 5.0 added another layer to this picture with the Slot-Based Execution (SBE) engine. Rather than processing queries as chains of document iterators, SBE compiles them into bytecode executed over named register slots—a virtual machine model that delivers significant throughput gains for pipelines containing `$group` or `$lookup` through batch processing. The limitation is that SBE has no concept of a work unit, which makes the Advanced/Works metric that FPTP relies on unavailable, forcing the optimizer to fall back to a selectivity-based score instead.

The limitations of traditional optimizer architectures—both cost-based and FFTP—have motivated an extensive body of research on learned query optimization, in which machine learning models are used to improve plan selection, cardinality estimation, and cost modeling. Harrison and Harrison [1] document a comprehensive methodology for MongoDB performance tuning spanning schema design, index selection, and query formulation. Rathore and Bagui [5] provide a broad architectural survey of MongoDB, while Nuriev et al. [6] analyze index optimization strategies in detail. At the intersection of machine learning and query optimization, a rich body of work—reviewed in Heinrich et al. [7]—examines learned cost models as alternatives to hand-crafted estimation functions. The prevailing paradigm, as synthesized by Panwar [8] and Karri and Muntala [9], involves replacing or augmenting the optimizer’s statistical components with data-driven models trained on historical query execution traces.

The present article investigates the query optimizer behavior of MongoDB 6.0.3 and 8.2.5 when executing aggregation pipelines, with particular focus on the FFTP mechanism and its interaction with the Pipeline Optimizer’s rule-based rewriting. Nine aggregation query types are systematically analyzed over a synthetic e-commerce order collection of 50K documents. For each query, all semantically valid operator permutations are examined together with their physical execution plans, enabling identification of three categories of cases in which the rule-based optimizer is demonstrably insufficient: IXSCAN preference bias at low selectivity, unbounded document multiplication following \$unwind, and failure to account for \$group output cardinality. Scalability is further assessed at 150K and 250K documents, confirming that index-provided sort with early termination yields constant execution time regardless of collection size, while \$unwind and full-scan patterns scale approximately linearly with collection size. A reproducible performance regression is observed for unbounded \$project operations in MongoDB 8.2.5 across all tested collection sizes, while most query types show improved performance compared to 6.0.3.

The objective of this study is to empirically characterize the behavior of MongoDB’s FFTP plan selection mechanism and Pipeline Optimizer across nine aggregation pipeline patterns, identifying the conditions under which the rule-based optimizer produces suboptimal results and quantifying the performance impact across versions and collection scales.

The study is organized around two orthogonal optimization mechanisms—FFTP plan selection and Pipeline Optimizer rewriting—whose interaction determines aggregation pipeline execution in MongoDB. The Pipeline Optimizer decides how the pipeline is structured, while FFTP decides which index is used to execute it. This separation is the central analytical lens of the study, formalized in Section 3.1, and the nine query types are designed to systematically probe its boundary conditions.

The remainder of the paper is organized as follows. Section 2 reviews related work on MongoDB query optimization. Section 3 describes the experimental environment, data collection methodology, and the baseline behavior of the Pipeline Optimizer. Section 4 concludes the article.

2. Related Work

Research directly relevant to this work spans two principal domains: query optimization in MongoDB, and query optimization in NoSQL systems more broadly. The closest prior work to this paper is the study by Tao et al. [4], which provides the first systematic experimental analysis of MongoDB’s FFTP optimizer. Published at the Australasian Database Conference (ADC 2024) and included in *Lecture Notes in Computer Science* (Springer), the paper shows that FFTP in MongoDB 7.0.1 has a built-in bias toward index scans—in practice, collection scans are often left out of the candidate set entirely, even when a full scan would be the faster choice. The authors attribute this to two concrete problems:

collection scans are frequently not generated as candidates at all, and the Advanced/Works metric underestimates the true cost of index access because it does not account for the additional step of fetching the actual document after locating the index key. When both issues are fixed—adding collection scans back as candidates and adjusting the productivity metric—plan selection improves noticeably. That said, the study stays within a narrow scope: simple conjunctive range queries with two predicates inside `find()`, on a single collection. Aggregation pipelines are explicitly left out, with the authors noting these as open problems.

Nuriev et al. [6] focus specifically on indexing in MongoDB—which fields to index, how to choose between B-tree, geospatial, text, and compound index types, and how to keep index size in check relative to working memory. What makes their work particularly relevant here is their use of `explain()` output and query profiler statistics to diagnose execution plans, which is the same diagnostic approach adopted in the present paper. Rathore and Bagui [5] look at MongoDB's architecture, covering the trade-offs that come with a document-oriented NoSQL design, while Chandra [10] examines the BASE consistency model that sits behind most NoSQL design decisions. Harrison and Harrison [1] take a more practical angle—their treatment of MongoDB performance tuning covers schema design, aggregation pipeline construction, and index selection, placing the optimization problem addressed in this paper into the broader context of application-level performance work.

The NoSQL optimization literature extends well beyond MongoDB. Karras et al. [11] tackle geospatial queries with an enhanced localized R-tree index designed for NoSQL systems. Mouhiha and Mabrouk [12] compare column-oriented NoSQL data warehouse models, testing clustering-based techniques for speeding up analytical queries. Chava et al. [13] take a wider view, surveying query optimization across document, key-value, column-family, and graph databases, with a focus on indexing, data partitioning, and caching under large-scale workloads. Seethala [14] brings machine learning into the picture, showing that predictive cost modeling and adaptive index selection can reduce query latency in both SQL and NoSQL systems, with examples drawn from financial and healthcare applications.

A decade ago, applying machine learning to query optimization was a niche idea. Today it represents one of the most active research directions in the database community. This shift was driven by a fundamental observation: traditional optimizers fail because their cardinality estimates are incorrect, often by orders of magnitude—a point demonstrated rigorously and with lasting impact by Leis et al. [3]. Early systems attempted to replace the optimizer entirely from scratch. Marcus et al. [15] built Neo around reinforcement learning, letting the optimizer learn iteratively from its own mistakes. The results were promising, but full replacement of a production optimizer is a hard sell. The Bao optimizer [16] offered a middle ground: keep the existing optimizer but use a learned model to steer it with per-query hints, updating continuously via Thompson sampling as workloads change. AutoSteer [17] automated the hint discovery process and showed 40% gains on real PrestoDB workloads. Kepler [18] went in a different direction entirely—skip cost estimation, train a model to predict the winning plan directly from parameter values, and fall back to the default optimizer when the model is uncertain.

Subsequent work pushed into more specialized territory. LOGER [2] and LEON [19] query optimization techniques explored reinforcement learning and cost model calibration, respectively. GenJoin [20] reframed plan selection as a generative problem. For queries that execute thousands of times, Tao et al. [21] and Marcus [22] argued that investing more time in finding a near-optimal plan upfront is worthwhile. Van De Water et al. [23] addressed a problem that is often overlooked—the origin and quality of training data for

learned optimizers. Negi et al. [24] investigated the robustness of learned optimizers under workload drift, when the model is no longer reliable. OptimAIzerSQL [25] combined a heuristic join order agent with a learned index selection component, a practical illustration that rule-based and learned approaches tend to complement rather than replace each other.

In contrast to this body of work, which is almost exclusively concerned with relational SQL systems employing cost-based optimizers, the present paper addresses a fundamentally different setting: the FFTP optimizer of MongoDB operating on aggregation pipelines in a document-oriented NoSQL context. To the best of our knowledge, no prior work systematically characterizes the failure modes of the FFTP mechanism for multi-stage aggregation queries, nor empirically quantifies the performance impact of operator ordering across MongoDB versions and collection scales.

The scientific contribution of this study is fourfold and can be summarized as follows. First, it provides the first systematic characterization of FFTP behavior across nine aggregation pipeline patterns, identifying the precise conditions under which the mechanism activates, selects correctly, or fails to differentiate between candidate plans. Second, it identifies and empirically quantifies three concrete failure modes of the rule-based optimizer: IXSCAN preference bias at low predicate selectivity, unbounded document multiplication following unwind, and inability to account for \$group output cardinality—each with measurable performance consequences. Third, it extends the analysis of Tao et al. [4], who examined FFTP exclusively within simple find() queries in MongoDB 7.0.1, to multi-stage aggregation pipelines across two major versions (6.0.3 and 8.2.5) and three collection scales (50K, 150K, and 250K documents). Fourth, it confirms a reproducible performance regression in unbounded \$match+ project queries in MongoDB 8.2.5 that cannot be attributed to a change in execution plan, establishing it as an open problem for future investigation.

3. Experiments

3.1. Conceptual Framework and Experimental Design Rationale

The experimental design is structured around two orthogonal optimization mechanisms that jointly determine aggregation pipeline execution in MongoDB. The first, FFTP plan selection, operates at the physical access path level: candidate plans are executed in parallel during a bounded trial period, and the plan that maximizes the Advanced/Works productivity ratio—or Score alone when the SBE engine is active—is selected and cached. The second, the Pipeline Optimizer, operates at the logical operator level prior to FFTP invocation, rewriting the user-defined pipeline through three deterministic, statistics-independent rules: predicate pushdown, index-provided sort elimination, and early termination propagation. Because the Pipeline Optimizer applies before FFTP and neither mechanism has visibility into the decisions of the other, their interaction can produce suboptimal outcomes that neither mechanism would generate in isolation.

To make this interaction concrete, consider a simple analogy. Rather than predicting which runner will win a race based on training records—as a cost-based optimizer does by estimating plan costs from statistics—MongoDB allows all candidates to race for a short distance and selects the one that covered the most ground with the least effort. This is the essence of FFTP: it measures actual runtime behavior instead of estimating it. The limitation is that a short trial may not reflect full-race performance—a runner who sprints well over 100 m may not be the best choice for a marathon. This is precisely the class of failure modes that the present study investigates.

Let $P = \langle s_1, s_2, \dots, s_n \rangle$ denote a user-defined aggregation pipeline of n stages. The Pipeline Optimizer applies a deterministic rewriting function $R: P \rightarrow P'$, producing an optimized pipeline $P' = \langle \$cursor(C), s_{k+1}, \dots, s_n \rangle$, where $\$cursor(C)$ absorbs the cursor layer stages $C = \langle s_1, \dots, s_k \rangle$ (i.e., those satisfiable through the storage engine),

and the remaining stages $s_{k+1}, \dots, s_n$ constitute the non-cursor layer invisible to FFTP. The rewriting R is rule-based and statistics-independent: it applies predicate pushdown, index-provided sort elimination, and early termination propagation unconditionally.

FFTP plan selection operates on the cursor layer only. Given a set of candidate plans $\{\pi_1, \pi_2, \dots, \pi_m\}$ generated from available indexes, each plan π_i is executed in parallel for a bounded trial period of W work units. The productivity of each plan is measured as:

$$\text{Score}(\pi_i) = \text{Advanced}(\pi_i) / \text{Works}(\pi_i) + \varepsilon, \quad (1)$$

where $\text{Advanced}(\pi_i)$ is the number of documents returned, $\text{Works}(\pi_i)$ is the total work performed, and ε is a small constant. The winning plan $\pi^* = \text{argmax } \text{Score}(\pi_i)$ is selected and cached. When the SBE engine is active (e.g., for \$group or \$lookup pipelines), Works is unavailable and Score is computed directly by the virtual machine without the Advanced/Works ratio.

This formalization makes explicit the orthogonality of R and FFTP: R operates at the logical operator level prior to FFTP invocation, while FFTP operates at the physical access path level within \$cursor(C) only. Neither mechanism has visibility into the decisions of the other, which is the root cause of the three failure modes identified in this study.

The two mechanisms operate on distinct layers of the pipeline, as illustrated in Figure 1.

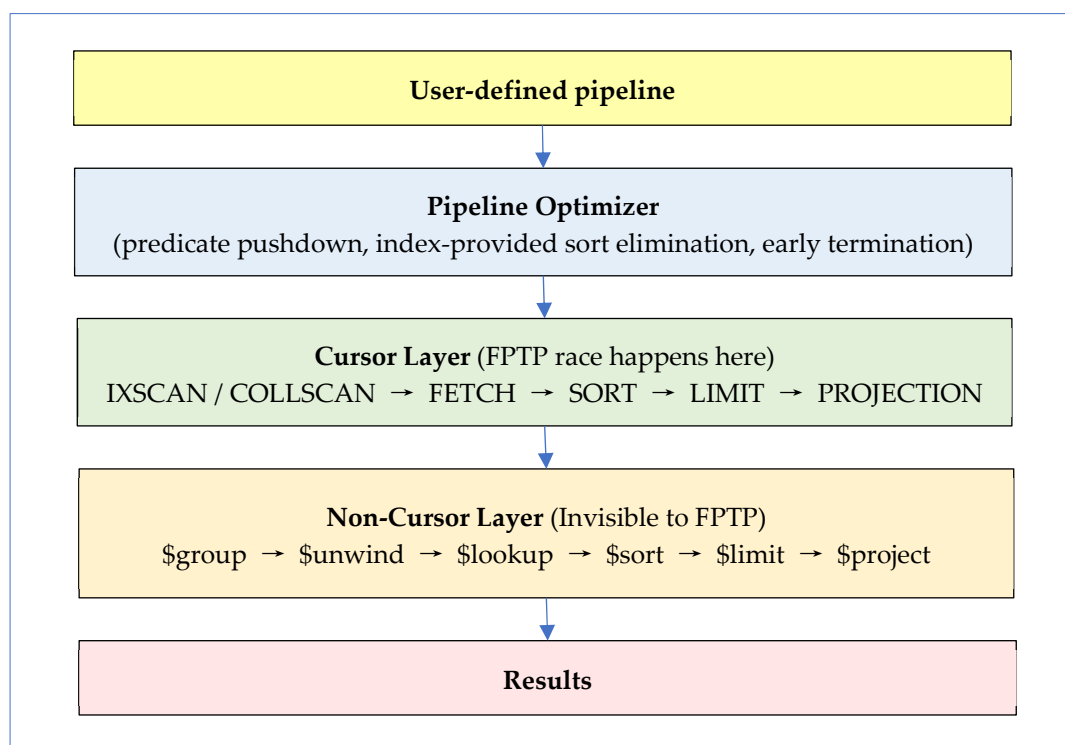


Figure 1. MongoDB aggregation pipeline execution layers.

In Figure 1, the following abbreviations are used: FFTP—First Past the Post (trial-based plan selection); IXSCAN—index scan; COLLSCAN—collection scan (full scan of all documents); FETCH—document retrieval from the storage engine; SORT—in-memory sort node; LIMIT—early termination after a fixed document count; PROJECTION—field selection applied at the cursor layer.

The cursor layer contains physical operators that execute directly against the storage engine—IXSCAN or COLLSCAN, FETCH, and optionally SORT, LIMIT, and PROJECTION when these can be satisfied through an index. Only operators in this layer participate in the FFTP race. The non-cursor layer contains the remaining pipeline stages—\$group, \$unwind,

\$lookup, and any \$sort or \$limit that cannot be pushed into the index scan—which execute after the cursor stage on the document stream it produces and are entirely invisible to the FFTP mechanism. This split is consequential: the plan selection decision is made exclusively based on what happens in the cursor layer, while operators in the non-cursor layer—regardless of how significantly they affect the result—exert no influence on which plan is chosen.

The nine query types are designed to systematically probe the boundary conditions of both mechanisms across five behavioral dimensions. The first dimension—FFTP activation and correct plan differentiation under competing indexes—is addressed by Tests 2, 3, 4, 6, 8, and 9, all of which involve multiple candidate plans competing during the trial period. The second dimension—FFTP behavior when blocking operators render Advanced = 0 for all candidate plans—is targeted by Tests 3 and 8, which contain \$group operators that prevent any results from being returned during the trial period. The third dimension—index-provided sort with early termination and its effect on execution time scaling—is examined by Tests 4, 6, and 9, where the winning plan eliminates the physical SORT node entirely and stops processing after a fixed number of results regardless of collection size. The fourth dimension—ESR compound index design and its interaction with the Pipeline Optimizer—is specifically isolated in Test 6, which empirically verifies that an ESR-ordered compound index simultaneously delivers predicate filtering and index-provided sort. The fifth dimension—FFTP preference bias at low predicate selectivity—is examined by Test 7, where the combined selectivity of the query predicates is practically equivalent to a full collection scan, yet the optimizer still prefers an index scan. Test 1 and Test 5 serve as baseline cases in which FFTP does not activate due to the absence of competing candidate plans. Test 1 confirms that plan competition requires at least two candidate plans; Test 5 additionally illustrates the document stream multiplication effect of \$unwind, which expands the input to \$group faster than the collection grows and represents a structural limitation that lies outside the scope of the FFTP mechanism entirely.

Each query type targets a distinct region of this design space, and together they cover the full range of Pipeline Optimizer transformations observed in practice. The metrics—Advanced, Works, and Score—are recorded directly from MongoDB’s explain (“allPlansExecution”) output and map unambiguously onto the FFTP productivity model described above, ensuring that the measured values can be interpreted within the same theoretical framework.

3.2. Database Description

3.2.1. Document Structure

The experiments use a synthetic orders collection in a database named fftp_analysis_db, deployed in MongoDB 6.0.3 and 8.2.5 on a local server. The collection comprises 50K documents with uniform distribution across all categorical fields. Table 1 describes the structure of the documents in the orders collection, the field names and their types.

The field’s product, category, and status are stored as textual string values with fixed finite value sets and uniform distribution across the collection—each of the five values per field covers approximately 20% of the documents. MongoDB’s schemaless document model imposes no typed constraints on these fields; string equality predicates in them are evaluated directly against the index keys, which store the string values verbatim.

Table 1. Document structure of the orders collection.

Field	Type	Description/Values
order_id	String	Unique identifier: ORD-0000001 ... ORD-0050000
customer_id	String	5000 distinct customers: CUST-00001 ... CUST-05000
product	String	Laptop, Phone, Tablet, Monitor, Keyboard
category	String	Electronics, Clothing, Food, Sports, Books
status	String	shipped, delivered, canceled, pending, processing
region	String	US, EU, ASIA, LATAM
quantity	Integer	Unit count, uniform 1–50
unit_price	Double	Unit price, uniform 10.00–2000.00
discount	Double	Discount rate, uniform 0.00–0.30
total_amount	Double	$\text{total_amount} = \text{quantity} \times \text{unit_price} \times (1 - \text{discount})$
order_date	Date	1 January 2022–31 December 2023
tags	Array	0–3 elements drawn from a fixed tag vocabulary

3.2.2. Index Configuration

Eight indexes are defined to support the query patterns used across all nine tests, as listed in Table 2.

Table 2. Indexes used.

Index	Key	Type
idx_status	{ status: 1 }	Single-field, ascending
idx_region	{ region: 1 }	Single-field, ascending
idx_category	{ category: 1 }	Single-field, ascending
idx_status_region	{ status: 1, region: 1 }	Compound, ESR-compatible
idx_total_amount	{ total_amount: -1 }	Single-field, descending
idx_order_date	{ order_date: 1 }	Single-field, range queries
idx_customer_id	{ customer_id: 1 }	Single-field
idx_cat_status_amount	{ category: 1, status: 1, total_amount: -1 }	Compound, ESR-covering

In the key definition column, 1 denotes an ascending index on the corresponding field and −1 denotes a descending index. For compound indexes, the field order is significant: fields used in equality predicates are placed first, fields used for sorting are placed next, and fields used in range predicates are placed last. This ordering—known as the ESR (Equality–Sort–Range) rule—allows a single compound index to simultaneously narrow the scan to a specific data segment and provide index-provided sort, eliminating the need for a physical sort node.

3.3. Metrics

Three metrics are recorded for each candidate plan during the FPTP trial period and form the basis of all subsequent analysis.

Advanced denotes the number of documents successfully returned by a plan during the trial period. In the document-iterator model used by the classic execution engine, each iteration either returns a result (advanced), requests more processing without producing a result (needTime), or signals that execution is complete (isEOF). A plan that returns many documents relative to the work it performs is considered highly productive.

Works denotes the total number of work units performed by the plan. By definition, $\text{Works} \geq \text{Advanced}$ in all cases. The difference between the two represents needTime steps—for instance, scanning index keys that do not match the query predicate.

Score is an internal FPTP productivity metric. In the classic engine, $\text{Score} \approx \text{Advanced} / \text{Works} + \text{constant}$. In the SBE engine, which handles pipelines containing \$group

or \$lookup, the virtual machine does not track work units, making Score the only reliable basis for comparing candidate plans.

One important consequence of this model is worth noting. Operators like \$group and \$sort without index-provided sort are blocking—they cannot return any results until they have processed all input documents. In a pipeline that contains \$group, Advanced equals zero for every candidate plan throughout the trial period. In this situation, FFTP has no productivity signal to work with and must differentiate between plans using Score alone.

3.4. Experimental Results

All results reported in this article were collected using two custom-built Python scripts. The ftp_analyzer.py script (Supplementary Materials/Code/ftp_analyzer.py) interfaces with MongoDB's explain() command in two verbosity modes: explain("queryPlanner") captures the Pipeline Optimizer's rewriting output—the optimized operator order and the stages absorbed into \$cursor—while explain("allPlansExecution") executes all candidate plans in parallel during the trial period and returns the FFTP metrics for each: Advanced, Works, and Score.

Execution time for the scalability analysis is measured separately using a dedicated Python script, which runs each pipeline 20 times with planCacheClear before every run and reports the median (Supplementary Materials/Code/benchmark.py). All raw results are serialized to JSON files for offline analysis (Supplementary Materials/DumpFiles/ *.json).

All tests are conducted under controlled conditions that are identical across both database versions: 50K documents, the same collection schema, the same eight indexes, and the same nine pipeline definitions. The sole variable is the MongoDB version—6.0.3 as the baseline and 8.2.5 as the current production release at the time of writing (February 2026). The hardware configuration, software versions, and technology stack used throughout all experiments are described in detail in Section 3.5.1.

The nine tests are designed to cover a representative range of aggregation pipeline patterns, from simple single-operator queries to complex multi-stage pipelines involving grouping, unwinding, and sorting. Each test targets a specific aspect of optimizer behavior: plan selection under competing indexes, interaction with the SBE engine, index-provided sort and early termination, ESR index design, and preference bias at low selectivity. The tests are presented in order of increasing complexity, and for each one the query definition, FFTP race results for both versions, and a comparative discussion are provided.

The ftp_analyzer.py script displays information on the console for each query regarding the logical plan and the selected physical plan, as well as statistical information for all analyzed plans (Supplementary Materials/Figure S1).

Test 1: Simple \$match on status with \$count

This test employs a straightforward pipeline that filters orders by status and counts the matching results. The query is shown in Listing 1. The goal is to determine whether FFTP activates when only one applicable index exists and the pipeline ends with \$count.

Listing 1. Simple \$match on status with \$count.

```
[
  { $match:{status:'shipped'} },
  { $count:'total' }
]
```

This query counts the total number of orders whose status is shipped. It is equivalent to the following SQL statement:

```
SELECT COUNT(*) AS total
```

```
FROM orders
WHERE status = 'shipped'.
```

In both versions, FFTP does not activate. There are no competing plans to race. The more interesting finding is the difference regarding how the two versions execute the query. MongoDB 6.0.3 uses a specialized COUNT_SCAN operator that reads index keys directly without loading the underlying documents—0 documents examined, 12,523 keys scanned, 4 ms. In MongoDB 8.2.5 the winning stage is GROUP, executed by the SBE engine, also at 4 ms. This does not represent a regression—it reflects how the SBE engine implements \$count in the 8.x versions through a more general aggregation mechanism rather than a dedicated operator. The result is the same, but the implementation method is less specialized. The conclusion is that when only a single index is applicable and no competing plans exist, FFTP has no basis for competition and is not activated.

Test 2: \$match on status + region (compound index)

This test examines FFTP behavior when a compound index (idx_status_region) competes against two single-field indexes during the trial period. The hypothesis is that the compound index will dominate, as it covers both predicates in a single scan without a post-filter step. The query is shown in Listing 2.

Listing 2. \$match on status and region with \$project.

```
[
  { $match:{status:'delivered',region:'EU'} },
  { $project:{order_id,total_amount} }
]
```

This query retrieves the order identifier and total amount for all orders that have been delivered to the EU region. It is equivalent to the following SQL statement:

```
SELECT order_id, total_amount
FROM orders
WHERE status = 'delivered' AND region = 'EU'.
```

The results are summarized in Table 3.

Table 3. Test 2—FFTP race results (three plans).

Plan	Strategy	Adv 6.0.3	Adv 8.2.5	Works 6.0.3	Works 8.2.5	Score 6.0.3	Score 8.2.5	Result
0	IXSCAN[idx_status_region] → FETCH → PROJECTION_SIMPLE	101	101	101	101	2.0002	2.0002	Winner (6.0.3 and 8.2.5)
1	IXSCAN[idx_region] → FETCH → PROJECTION_SIMPLE	18	29	101	101	1.1784	1.2873	Rejected
2	IXSCAN[idx_status] → FETCH → PROJECTION_SIMPLE	19	25	101	101	1.1883	1.2477	Rejected

In Tables 3–5, Adv denotes the number of documents returned (Advanced) by each candidate plan during the FFTP (First Past the Post) trial period; Works denotes the total work units performed; Score is the FFTP productivity metric (see Section 3.3).

Table 4. Test 3—FPTP race results (two plans).

Plan	Strategy	Adv 6.0.3	Adv 8.2.5	Works 6.0.3	Works 8.2.5	Score 6.0.3	Score 8.2.5	Result
0	IXSCAN[idx_category] → FETCH → GROUP	0	101	0	101	2.0002	2.0002	Winner (6.0.3 and 8.2.5)
1	IXSCAN[idx_cat_status_amount] → FETCH → PROJECTION_SIMPLE	0	101	0	101	2.0002	2.0002	Rejected

Table 5. Test 4—FPTP race results (three plans).

Plan	Strategy	Adv 6.0.3	Adv 8.2.5	Works 6.0.3	Works 8.2.5	Score 6.0.3	Score 8.2.5	Results
0	IXSCAN[idx_total_amount] → FETCH → PROJECTION_SIMPLE → LIMIT	100	100	N/A	188	2.5319	2.5321	Winner (6.0.3 and 8.2.5)
1	IXSCAN[idx_status] → FETCH → SORT → PROJECTION_SIMPLE	0	0	N/A	188	1.0001	1.0001	Rejected
2	IXSCAN[idx_status_region] → FETCH → SORT → PROJECTION_SIMPLE	0	0	N/A	188	1.0001	1.0001	Rejected

FPTP activates in both versions with three competing plans, and the winning strategy is identical in both—execution time is 6.5 ms in 6.0.3 and 8 ms in 8.2.5, a 23% slowdown in the newer version. Plan 0 achieves Advanced = Works = 101, meaning every index key scanned maps directly to a result document with no wasted work. The single-field indexes, by contrast, require a FETCH followed by a post-filter on the second predicate, with only 19–25% of fetched documents passing the filter—hence the substantially lower Advanced counts. FPTP correctly identifies the compound index as the dominant plan in both versions.

Test 3: \$match → \$group (aggregation by category, SBE engine)

This test examines how FPTP behaves when the pipeline includes a \$group operator and execution is handled by the SBE engine. The query is shown in Listing 3.

Listing 3. \$match on category followed by \$group and \$sort.

```
[
  { $match:{category:'Electronics'} },
  { $group:{_id:'$status',total_revenue:$sum, avg_quantity:{$avg},count:{$sum:1}}
}, { $sort:{total_revenue:-1} }
]
```

This query filters orders in the “Electronics” category, then groups them by status, computing for each group the total revenue, average quantity, and order count. The results are sorted by total revenue in descending order. It is equivalent to the following SQL statement:

```
SELECT status,
       SUM(total_amount) AS total_revenue,
       AVG(quantity)     AS avg_quantity,
```

```

COUNT(*)      AS count
FROM orders
WHERE category = 'Electronics'
GROUP BY status
ORDER BY total_revenue DESC.

```

The results are summarized in Table 4.

Test 3 exposes a structural limitation that remains unresolved in MongoDB 8.2.5. Both candidate plans produce an identical Score of 2.0002 in both versions—the winner (`idx_category`) is selected by its position in the plan array, not because it outperforms the alternative on any measurable metric. The key difference between versions is in the availability of Works: in 6.0.3, Works = 0 because the SBE engine does not track work units, and the blocking nature of `$group` means Advanced = 0 throughout the trial period; in 8.2.5, Works = 101 is available. Even so, this changes nothing—both plans process the same number of documents, and the Score remains numerically indistinguishable. Execution time improves from 42 ms in 6.0.3 to 31 ms in 8.2.5, a 26% gain. Despite `idx_cat_status_amount` being the theoretically superior choice—it covers both category and status—FPTP cannot identify it as such when scores are tied. The limitation is structural: resolving it would require statistics on `$group` output cardinality, which the FPTP mechanism does not have access to.

Test 4: `$match` → `$sort` → `$limit` (Top-N query)

This test examines FPTP behavior in a Top-N retrieval query. The hypothesis is that `idx_total_amount`, being a descending index, provides index-provided sort and enables early termination after 100 documents have been collected. The query is shown in Listing 4.

Listing 4. Top-N query with `$match`, `$sort`, `$limit`, and `$project`.

```

[
  { $match:{status:{$in:['shipped','delivered']}} },
  { $sort:{total_amount:-1} },
  { $limit:100 },
  { $project:{order_id,total_amount,region} }
]

```

This query retrieves the top 100 highest-value orders among those that have been shipped or delivered, returning only the order identifier, total amount, and region. It is equivalent to the following SQL statement:

```

SELECT order_id, total_amount, region
FROM orders
WHERE status IN ('shipped', 'delivered')
ORDER BY total_amount DESC
FETCH FIRST 100 ROWS ONLY.

```

The results are summarized in Table 5.

Plan 0 wins decisively in both versions. `idx_total_amount` is a descending index—MongoDB traverses keys in descending order and stops as soon as 100 documents have been collected. The 188 keys examined break down as 100 results plus 88 documents that did not pass the `$match` filter. Plans 1 and 2 have no such advantage: they require an in-memory sort of roughly 25,000 matching documents before `$limit` can be applied, which means Advanced = 0 throughout the trial period. A Score of 2.5321 against 1.0001 represents a decisive margin. Execution time is 2 ms in both versions. The Pipeline Optimizer applies the same transformation in both versions, collapsing four logical stages into a single `$cursor` with index-provided sort (*).

Test 5: \$match → \$unwind → \$group → \$sort → \$limit

This test examines FFTP behavior on a more complex pipeline that combines a date range filter with \$unwind and \$group. The query is shown in Listing 5.

Listing 5. Complex pipeline with \$unwind and \$group over a date range.

```
[
  { $match:{order_date:{$gte:ISODate('2023-01-01')}} },
  { $unwind:'$tags' },
  { $group:{_id:{tag,region},revenue,orders} },
  { $sort:{revenue:-1} },
  { $limit:20 }
]
```

This query filters orders placed on or after 1 January 2023, then unnests the tags array so that each tag value becomes a separate row, groups the results by tag and region (computing total revenue and order count per group), and returns the top 20 groups by revenue. The \$unwind operator has no direct SQL equivalent in standard SQL, but can be approximated using UNNEST or LATERAL JOIN in systems that support these constructs, such as PostgreSQL:

```
SELECT t.tag, o.region,
       SUM(o.total_amount) AS revenue,
       COUNT(*)           AS orders
FROM orders o,
     UNNEST(o.tags) AS t(tag)
WHERE o.order_date >= '2023-01-01'
GROUP BY t.tag, o.region
ORDER BY revenue DESC
FETCH FIRST 20 ROWS ONLY.
```

The results are summarized in Table 6.

Table 6. Test 5—results for MongoDB 6.0.3 and 8.2.5.

Metric	MongoDB 6.0.3	MongoDB 8.2.5	Difference
FFTP active	No (0 plans)	No (0 plans)	None
Winning Index	idx_order_date	idx_order_date	None
Execution Time	131 ms	103 ms	21% improvement in 8.2.5
Pipeline Optimizer	5 → 4 stages	5 → 4 stages	None
FFTP active	No (0 plans)	No (0 plans)	None

FFTP does not activate in either version—only one index is applicable to the date range predicate, leaving no competing plans to race. After \$match, 25,045 documents pass the filter; after \$unwind, the document count expands to approximately 38,000, reflecting an average of around 1.5 tags per document. The final output is 20 documents. The Winning Plan Stage is PROJECTION_SIMPLE. \$cursor absorbs \$match + \$project. The primary source of latency is \$unwind, which expands the tags array and multiplies the document stream before it reaches \$group. The \$sort and \$limit stages that follow \$group cannot be reordered by the Pipeline Optimizer, as \$group is a blocking operator whose output bears no relationship to any index ordering.

Test 6: Optimal compound index (ESR rule)

This test provides an empirical verification of the Equality → Sort → Range (ESR) rule. The rule specifies the recommended field order for compound indexes in MongoDB:

equality predicates first, the sort field next, and range predicates last. An index following this order can simultaneously narrow the scan to a specific data segment and provide index-provided sort, eliminating the need for a physical sort node. The hypothesis is that `idx_cat_status_amount = { category: 1, status: 1, total_amount: -1 }` will dominate by virtue of index-provided sort. The query is shown in Listing 6.

Listing 6. ESR-compliant query with \$match, \$sort, \$limit, and \$project.

```
[
  { $match:{category:'Electronics', status:'delivered'} },
  { $sort:{total_amount:-1} },
  { $limit:50 },
  { $project:{product,total_amount,region} }
]
```

This query retrieves the top 50 highest-value delivered orders in the Electronics category, returning only the product name, total amount, and region. It is equivalent to the following SQL statement:

```
SELECT product, total_amount, region
FROM orders
WHERE category = 'Electronics'
      AND status = 'delivered'
ORDER BY total_amount DESC
FETCH FIRST 50 ROWS ONLY.
```

The results are summarized in Table 7.

Table 7. Test 6—FPTP race results (four plans).

Plan	Strategy	Adv 6.0.3	Adv 8.2.5	Works 6.0.3	Works 8.2.5	Score 6.0.3	Score 8.2.5	Result
0	IXSCAN[idx_cat_status_amount] → FETCH → PROJECTION_SIMPLE → LIMIT	50	50	50	50	3.0002	3.0002	Winner (6.0.3 and 8.2.5)
1	IXSCAN[idx_status] → FETCH → SORT → PROJECTION_SIMPLE	0	0	50	50	1.0001	1.0001	Rejected
2	IXSCAN[idx_category] → FETCH → SORT → PROJECTION_SIMPLE	0	0	50	50	1.0001	1.0001	Rejected
3	IXSCAN[idx_status_region] → FETCH → SORT → PROJECTION_SIMPLE	0	0	50	50	1.0001	1.0001	Rejected

Test 6 produces the most consistent result in the study: 1 ms in both versions without exception. A Score of 3.0002, where Advanced = Works = 50, is the highest value observed across all nine tests—three times the score of any competing plan—and is identical in both versions. The ESR field order delivers two things at once: the equality fields (category, status) restrict the scan to a narrow data segment, while the sort field (total_amount DESC) provides index-provided sort and allows early termination after exactly 50 documents. The ESR principle for compound index design is confirmed to be version-independent.

Test 7: IXSCAN preference bias

This test examines plan selection behavior when the combined selectivity of the query predicates is low. The query is shown in Listing 7.

Listing 7. Low selectivity \$match with \$group aggregation.

```
[
  { $match:{discount:{>0.2},product:{<ne:'Laptop'},
    total_amount:{>=500}} }, { $group:{_id:'$region',count,avg} }
]
```

This query filters orders with a discount greater than 20%, excluding Laptop products, and a total amount of at least 500, then groups the results by region computing the order count and average total amount per region. The combination of three predicates with low combined selectivity makes this query particularly relevant for studying optimizer behavior. It is equivalent to the following SQL statement:

```
SELECT region,
       COUNT(*)           AS count,
       AVG(total_amount)  AS avg
FROM orders
WHERE discount > 0.2
      AND product <> 'Laptop'
      AND total_amount >= 500
GROUP BY region.
```

The results are summarized in Table 8.

Table 8. Test 7—results for MongoDB 6.0.3 and 8.2.5.

Metric	MongoDB 6.0.3	MongoDB 8.2.5	Interpretation
Winning Stage	GROUP	GROUP	Identical
Winning Index	COLLSCAN	idx_total_amount	Critical difference
Physical plan	COLLSCAN → GROUP	IXSCAN → FETCH → GROUP	Different strategy
Index Filter Ratio	0.0001	0.0001	Identical selectivity
Execution Time	163 ms	134 ms	8.2.5 is faster but selects the suboptimal plan

FPTP does not activate in either version—no competing plans are generated. The noteworthy finding here is how the two versions differ in their plan choice. MongoDB 6.0.3 selects COLLSCAN at 163 ms—a reasonable decision given that an Index Filter Ratio of 0.0001 means the index covers almost no documents, and the mandatory FETCH for every index key renders IXSCAN more expensive than scanning the collection outright. MongoDB 8.2.5 selects IXSCAN[idx_total_amount] and runs in 134 ms—not because this is the superior choice, but because the specific combination of data distribution and hardware happens to favor it at 50K documents. The selected index does not even cover the \$match predicates, yet 8.2.5 still prefers it, scanning 48,895 out of 50,000 documents with the added overhead of index traversal.

The IXSCAN preference bias documented by Tao et al. [4] for MongoDB 7.0.1 remains present in 8.2.5—and under more adverse conditions than previously reported. Where 6.0.3 correctly reverts to COLLSCAN at ratio = 0.0001, 8.2.5 selects the index regardless. The bias remains unaddressed.

Test 8: \$match → \$group—canceled orders by customer (SBE engine)

This test analyzes canceled orders grouped by customers, with the SBE engine handling execution and a compound index competing against two single-field indexes. The query is shown in Listing 8.

Listing 8. Canceled orders analysis by customer with \$group, \$sort, and \$limit.

```
[
  { $match:{status:'cancelled',region:'US'} },
  { group:{_id:'$customer_id',cancelled:{$sum:1},lost_revenue:{$sum}} },
  { $sort:{lost_revenue:-1} },
  { $limit:10 }
]
```

This query identifies the top 10 customers in the US region with the highest lost revenue due to canceled orders, computing for each customer the number of cancelations and the total amount lost. It is equivalent to the following SQL statement:

```
SELECT customer_id,
       COUNT(*)      AS canceled,
       SUM(total_amount) AS lost_revenue
FROM orders
WHERE status = 'canceled'
      AND region = 'US'
GROUP BY customer_id
ORDER BY lost_revenue DESC
FETCH FIRST 10 ROWS ONLY.
```

The results are summarized in Table 9.

Table 9. Test 8—FPTP race results (three plans).

Plan	Strategy	Adv 6.0.3	Adv 8.2.5	Works 6.0.3	Works 8.2.5	Score 6.0.3	Score 8.2.5	Result
0	IXSCAN[idx_status_region] → FETCH → GROUP	N/A	101	0	101	1.5027	2.0002	Winner (6.0.3 and 8.2.5)
1	IXSCAN[idx_region] → FETCH → PROJECTION_SIMPLE	N/A	19	0	101	1.1445	1.1883	Rejected
2	IXSCAN[idx_status] → FETCH → PROJECTION_SIMPLE	N/A	19	0	101	1.1301	1.1883	Rejected

Unlike Test 3, where both plans scored identically and the winner was determined by position, FPTP correctly differentiates between the plans here. idx_status_region covers both \$match predicates and scans only 3166 documents, yielding an Index Filter Ratio of 0.7429. This translates into a decisive Score advantage: 2.0002 for Plan 0 against approximately 1.19 for the alternatives. As in Test 3, Works = 0 in 6.0.3 because the SBE engine does not track work units, while Works = 101 is available in 8.2.5. Execution time is 16.5 ms in 6.0.3 and 15 ms in 8.2.5, a 9% improvement.

Test 9: \$match → \$group—canceled orders by customer (SBE engine)

This test verifies the early termination effect when \$limit appears at position three out of four stages in the pipeline. The query is shown in Listing 9.

Listing 9. Early termination query with \$limit at position 3/4.

```
[
  { "$match": {"status": "shipped"} }
  { "$sort": {"total_amount": -1} },
  { "$limit": 10 },
  { "$project": {"order_id": 1, "total_amount": 1, "region": 1, "_id": 0} }
]
```

This query retrieves the top 10 highest-value shipped orders, returning only the order identifier, total amount, and region. It is structurally identical to T4 but with a much smaller limit value, making it specifically designed to test whether early termination via index-provided sort is effective when \$limit is placed at position three out of four pipeline stages. It is equivalent to the following SQL statement:

```
SELECT order_id, total_amount, region
FROM orders
WHERE status = 'shipped'
ORDER BY total_amount DESC
FETCH FIRST 10 ROWS ONLY.
```

The results are summarized in Table 10.

Table 10. Test 9—FPTP race results (three plans).

Plan	Strategy	Adv 6.0.3	Adv 8.2.5	Works 6.0.3	Works 8.2.5	Score 6.0.3	Score 8.2.5	Result
0	IXSCAN[idx_total_amount] → FETCH → PROJECTION_SIMPLE → LIMIT	10	10	25	36	2.4002	2.2780	Winner (6.0.3 and 8.2.5)
1	IXSCAN[idx_status] → FETCH → SORT → PROJECTION_SIMPLE	0	0	25	36	1.0001	1.0001	Rejected
2	IXSCAN[idx_status_region] → FETCH → SORT → PROJECTION_SIMPLE	0	0	25	36	1.0001	1.0001	Rejected

Test 9 demonstrates consistent FPTP behavior across both versions in an early termination scenario. Plan 0 delivers 10 documents during the trial period through index-provided sort, stopping as soon as the \$limit threshold is reached. Plans 1 and 2 require a physical sort node and cannot return any document until all matching documents have been sorted—Advanced = 0 throughout the trial period. The resulting Score gap—roughly 2.4 against 1.0001—follows the same pattern observed in Tests 4 and 6. The small numerical differences between versions—docsExamined = 25 in 6.0.3 versus 36 in 8.2.5, and Score = 2.4002 versus 2.2780—fall within the expected range given different random data distributions. The availability of Works in both versions confirms that Works = 0 in 6.0.3 is specific to the SBE engine and \$group pipelines, rather than a general limitation of the FPTP mechanism.

3.5. Summary of Results

Table 11 summarizes the experimental results across all nine tests. Execution times are medians measured at 50K documents.

Table 11. Summary of results across all nine tests.

Test	FPTP 6.0.3	FPTP 8.2.5	Plans 6.x/8.x	Time 6.0.3	Time 8.2.5	Δ Time *	Winning Stage, MongoDB 6.0.3	Winning Stage, MongoDB 8.2.5
T1	No	No	0/0	4 ms	4 ms	0%	COUNT_SCAN	GROUP
T2	Yes	Yes	3/3	6.5 ms	8 ms	+23%	PROJECTION_SIMPLE	PROJECTION_SIMPLE
T3	Yes	Yes	2/2	42 ms	31 ms	−26%	GROUP	GROUP
T4	Yes	Yes	3/3	2 ms	2 ms	0%	LIMIT	LIMIT
T5	No	No	0/0	130.5 ms	102.5 ms	−21%	PROJECTION_SIMPLE	PROJECTION_SIMPLE
T6	Yes	Yes	4/4	1 ms	1 ms	0%	LIMIT	LIMIT
T7	No	No	0/0	163 ms	133.5 ms	−18%	GROUP (COLLSCAN)	GROUP (IXSCAN)
T8	Yes	Yes	3/3	16.5 ms	15 ms	−9%	GROUP	GROUP
T9	Yes	Yes	3/3	1 ms	1 ms	0%	LIMIT	LIMIT

* Δ Time = (Time 8.2.5 − Time 6.0.3)/Time 6.0.3 × 100%. Positive values indicate regression; negative values indicate improvement.

FPTP activation is identical across both versions—six out of nine tests. It does not activate for T1, T5, and T7, where only a single candidate plan is generated. The Pipeline Optimizer applies identical transformations in both versions without exception. Under controlled conditions—identical hardware, median values over 20 repeated runs—MongoDB 8.2.5 shows improved performance in T3 (−26%), T5 (−21%), and T7 (−18%). The sole regression is T2 (+23%), discussed in Section 3.3. Queries supported by an ESR-ordered index are version-independent: T4, T6, and T9 remain stable at 1–2 ms in both versions.

3.5.1. Execution Time Analysis

All measurements are conducted on a single physical machine to eliminate hardware variability and ensure controlled conditions for comparing the two database versions. The machine (DELL laptop) runs Windows 10 Enterprise on an Intel Core i5-11320H (11th generation, 4 cores/8 threads, 3.20 GHz base clock, 4.50 GHz Turbo), with 16 GB DDR4 3200 MHz and a KIOXIA KBG40ZNS512G 512 GB NVMe SSD (PCIe Gen 3). The same hardware configuration is used for every test—processor, memory, storage device, and operating system remain unchanged throughout.

Both MongoDB versions—6.0.3 and 8.2.5—are installed in separate directories, each with its own data path, log path, and network port to prevent resource conflicts. The two instances never run concurrently: each version is started, tested, and shut down before the other is launched.

All experiments are implemented in Python 3.12.6 using the following technology stack. The MongoDB driver pymongo 4.16.0 is used for all database interactions, including query execution and explain() output collection. Terminal output formatting relies on colorama 0.4.6 and tabulate 0.9.0. Statistical computations—including median calculation across the 20 repeated runs—are performed using numpy 2.4.2. Execution time plots are generated with matplotlib 3.9.4. The complete dependency specification is provided in Supplementary Materials/Code/requirements.txt.

Tests are conducted at three collection sizes—50K, 150K, and 250K documents—using the same schema and index configuration throughout. Each version and size combination is executed 20 times, with two warmup runs and a planCacheClear before each measurement. Median values are adopted as the primary metric because of their robustness to outliers, particularly for sub-5 ms tests. The results are shown in Table 12.

Table 12. Median execution time (ms) by test, version, and collection size.

Test	MongoDB 6.0.3—Median (ms)			MongoDB 8.2.5—Median (ms)			Scaling
	50K	150K	250K	50K	150K	250K	
T1	4	15	23	4	13.5	18	Δ : 0%/−10%/−22%
T2	6.5	18	28	8	18.5	32	Δ : +23%/+3%/+14%
T3	42	151	237	31	108.5	199.5	Δ : −26%/−28%/−16%
T4	2	3	3	2	3.5	3.5	Version- and size-independent
T5	130.5	357	735.5	102.5	376	668.5	Δ : −21%/+5%/−9%
T6	1	2	2	1	2	2	Version- and size-independent
T7	163	423.5	740.5	133.5	405.5	699	Δ : −18%/−4%/−6%
T8	16.5	31	47	15	28	42	Δ : −9%/−10%/−11%
T9	1	1	1	1	2	1.5	Version- and size-independent

Table 12 was generated using a special Python script (Supplementary Materials/Code/plot_scaling.py) that reads the execution time for each plan across all versions

and collection sizes from the JSON benchmark files. This script also plots the median execution time by test, version, and collection size (Supplementary Materials/Figure S3).

The results fall into three distinct groups based on performance scales with collection size. T4, T6, and T9 exhibit constant execution time across all three sizes in both versions, remaining in the 1–3 ms range throughout. The underlying reason is identical in all three cases: index-provided sort with early termination caps the number of documents processed at a fixed count—100, 50, and 10 respectively—regardless of how large the collection grows. This confirms that appropriate index design combined with early termination yields behavior that is both version-independent and size-independent.

Tests T1, T2, T5, T7, and T8 scale approximately linearly with collection size in both versions. T5 (\$unwind + \$group) exhibits mildly superlinear scaling: execution time grows by roughly $5.6\times$ in MongoDB 6.0.3 (130.5 ms $\rightarrow$ 735.5 ms) and $6.5\times$ in MongoDB 8.2.5 (102.5 ms $\rightarrow$ 668.5 ms) for a $5\times$ increase in data volume. The additional growth stems from \$unwind multiplying the document stream before it reaches \$group, so the effective input to the aggregation stage grows faster than the collection itself. T7 (IXSCAN preference bias) scales at roughly $4.5\times$ for 6.0.3 (163 ms $\rightarrow$ 740.5 ms) and $5.2\times$ for 8.2.5 (133.5 ms $\rightarrow$ 699 ms)—close to linear in both cases, since a low-selectivity IXSCAN scans a proportionally larger share of the index as the collection grows.

T3 (\$match + \$group, SBE) exhibits moderate superlinear scaling in both versions, at approximately $5.6\times$ for 6.0.3 and $6.4\times$ for 8.2.5. MongoDB 8.2.5 is consistently faster than 6.0.3 for T3 at every collection size—by 26% at 50K, 28% at 150K, and 16% at 250K—likely reflecting improvements to the SBE engine’s \$group implementation between versions.

Across versions, MongoDB 8.2.5 outperforms 6.0.3 for T3 (−16% to −28% at all sizes), T5 (−21% at 50K, approximately equal at 150K and 250K), T7 (−18% at 50K, approximately equal at 150K and 250K), and T8 (−9% to −11%). The sole regression is T2 (\$match + \$project), which is reproducible at all three collection sizes (+23% at 50K, +3% at 150K, and +14% at 250K) despite an identical execution plan between versions. The root cause of this regression remains unverified without server-level profiling. T4 and T9 show minor fluctuations of ± 1 ms, which fall within the resolution of the timer.

3.5.2. Deployment Environment Comparison

To assess whether the findings obtained from the local machine generalize to a cloud deployment, all nine benchmark queries are executed in MongoDB Atlas under identical conditions—the same schema, the same eight indexes, the same nine pipeline definitions, and the same measurement methodology (20 runs with planCacheClear before each measurement, median values reported). The Atlas cluster runs MongoDB 8.0.20—a different minor version from the locally tested 8.2.5—and the comparison is therefore limited to behavioral patterns and measurement stability rather than version-specific performance characteristics. The median execution times on Atlas are comparable to those recorded on the local machine, confirming that the behavioral patterns identified in Section 3.5.1—constant scaling for index-provided sort queries, linear scaling for full-scan and aggregation queries, and mildly superlinear scaling for \$unwind + \$group queries—hold in a cloud deployment environment.

A notable difference between the two environments concerns measurement stability. Table 13 reports the coefficient of variation (cv = standard deviation/mean $\times$ 100%) for both environments across all three collection sizes.

Table 13. Coefficient of variation (cv, %) by test, environment, and collection size.

Test	50K Documents		150K Documents		250K Documents	
	Local	Atlas	Local	Atlas	Local	Atlas
T1	38.6	0.0	17.0	3.6	14.1	1.8
T2	21.9	6.1	12.6	2.4	23.1	1.6
T3	12.7	0.0	13.9	0.8	11.4	0.6
T4	38.2	0.0	34.7	0.0	26.1	0.0
T5	13.7	1.5	10.7	1.0	6.7	0.8
T6	44.3	0.0	38.9	0.0	43.3	0.0
T7	11.3	1.4	7.5	1.1	3.8	1.3
T8	14.9	2.5	7.7	1.8	14.1	2.5
T9	58.3	447.2 *	44.3	447.2 *	39.0	447.2 *

Except for T9, Atlas consistently exhibits substantially lower cv values than the local machine—on average 5 to 15 times lower across all tests and collection sizes. The most pronounced differences are observed for T1, T4, and T6, where cv reaches 38–44% locally yet drops to 0.0% in Atlas. For T4 and T6, the Atlas measurements are perfectly deterministic at every collection size—every run returns an identical execution time. This stability is attributable to the isolated infrastructure of the cloud deployment, which eliminates the OS-level scheduling interference and background process contention that affect measurements on a shared local machine. For computationally intensive queries—T3, T5, T7, and T8—the cv in Atlas remains below 3% at all collection sizes, compared to 7–15% locally.

The anomalous cv value for T9 in Atlas (447.2% at all three sizes, marked with * in Table 13) is an artifact of timer resolution rather than genuine measurement instability. Across all 20 runs at every collection size, 19 measurements record 0 ms and a single measurement records 1 ms. The median is 0 ms, indicating that the query consistently completes in under 1 ms—below the resolution of the timer. The elevated cv arises because the standard deviation formula amplifies a single 1 ms outlier when the mean is 0.05 ms, producing a mathematically large but practically meaningless coefficient.

These results suggest that median-based reporting, as adopted throughout this study, is a robust choice for the local environment. The behavioral conclusions of the study—plan selection outcomes, scaling categories, and version comparisons—are unaffected by the difference in measurement stability between environments, as they are derived from median values rather than individual runs.

3.6. MongoDB Pipeline Optimizer

3.6.1. Purpose and Operating Principle

The Pipeline Optimizer is a component of the MongoDB query optimizer that executes before the FFTP mechanism. Its purpose is to rewrite the user-defined aggregation pipeline into a more efficient form before execution begins. Where FFTP determines which index to use, the Pipeline Optimizer determines how to reorder and consolidate the operators themselves.

The result of the rewriting can be inspected through the following command:

```
db.collection.aggregate([...]).explain("queryPlanner").
```

For each aggregation query, the `fptr_analyzer.py` script outputs information to the console about the original and optimized order of the operators (Supplementary Materials/Figure S2).

The core principle is the division of the pipeline into two layers. The first is the \$cursor stage, which contains the physical operators that execute directly against the index—IXSCAN or COLLSCAN, FETCH, and optionally SORT, LIMIT, and PROJECTION when these can be satisfied through the index. The second layer holds the remaining pipeline stages that execute after the cursor stage—\$group, \$unwind, \$lookup, and any \$sort or \$limit that cannot be pushed into the index scan.

Three deterministic optimizations are applied. First, predicate pushdown: \$match is relocated as early as possible in the pipeline to reduce the number of documents that subsequent stages must process. Second, index-provided sort: if an available index delivers documents in the order required by \$sort, no physical SORT node is created and \$sort is absorbed into the \$cursor stage. Third, early termination: \$limit is pushed as far forward in the execution as possible, allowing MongoDB to cease processing documents as soon as the required count is reached.

3.6.2. Results

The Pipeline Optimizer behaves identically in MongoDB 6.0.3 and 8.2.5—every transformation is applied in an identical manner across both versions. Table 14 summarizes the optimizer’s behavior for all nine tests, showing the original operator order as defined by the user and the rewritten order after transformation. The symbol * indicates that \$sort has been eliminated through index-provided sort, meaning no physical SORT node is created.

Table 14. Pipeline Optimizer transformations across all nine tests.

Test	Original Pipeline	Optimized Pipeline	Optimization
T1	\$match → \$count	\$cursor(\$match + \$count)	COUNT_SCAN/GROUP
T2	\$match → \$project	\$cursor(\$match + \$project)	\$project absorbed into cursor
T3	\$match → \$group → \$sort	\$cursor(\$match + \$group) → \$sort	\$group in SBE cursor; \$sort remains
T4	\$match → \$sort → \$limit → \$project	\$cursor(\$match + \$sort* + \$limit + \$project)	Index-provided sort; 4 → 1 stages
T5	\$match → \$unwind → \$group → \$sort → \$limit	\$cursor(\$match) → \$unwind → \$group → \$sort → \$limit	No reordering
T6	\$match → \$sort → \$limit → \$project	\$cursor(\$match + \$sort* + \$limit + \$project)	Index-provided sort (ESR); 4→1 stages
T7	\$match → \$group	\$cursor(\$match + \$group)	\$group in SBE cursor
T8	\$match → \$group → \$sort → \$limit	\$cursor(\$match + \$group) → \$sort → \$limit	\$group in SBE cursor; \$sort and \$limit remain
T9	\$match → \$sort → \$limit → \$project	\$cursor(\$match + \$sort* + \$limit + \$project)	Index-provided sort; 4→1 stages

In Table 14, the following abbreviations are used: SBE—Slot-Based Execution engine; ESR—Equality–Sort–Range (recommended field ordering rule for compound indexes); *—\$sort is eliminated through index-provided sort; no physical SORT node is created.

Index-provided sort, observed in Tests 4, 6, and 9, is the most impactful optimization in the set—it eliminates the physical SORT node entirely and reduces the visible pipeline from four stages to one. It requires an exact correspondence between the \$sort field and direction and the corresponding index key. For SBE queries in Tests 3, 7, and 8, \$group executes inside the cursor stage, but any \$sort that follows \$group always remains as a separate stage—\$group is a blocking operation whose output order bears no relationship to any index. Test 5 is the only case in which the visible pipeline structure remains unchanged: \$unwind prevents any

reordering of the stages that follow it. The identical behavior of the Pipeline Optimizer across both versions confirms that its transformations are rule-based and static, independent of runtime data characteristics and of which execution engine is active.

Test 7 directly confirms the preference bias documented by Tao et al. [4]: at an Index Filter Ratio of 0.0001, MongoDB chooses IXSCAN over COLLSCAN and ends up scanning 48,895 out of 50,000 documents, with the added overhead of index traversal on top. What makes this observation particularly significant is that Tao et al. [4] identified and measured this bias in MongoDB 7.0.1, noting that it had already diminished compared to earlier versions. The present study finds it persistent and unchanged in 8.2.5.

3.6.3. Relationship with Relational Optimization Principles

The transformations listed in Table 14 warrant a systematic comparison with classical relational optimization. Several of the Pipeline Optimizer's rules have direct analogs in relational systems, while others arise exclusively from MongoDB's document-oriented model and FFTP architecture. Table 15 makes this distinction explicit, mapping each observed optimization against its relational analog and identifying what is MongoDB-specific.

Table 15. Pipeline Optimizer transformations: relational analogs vs. MongoDB-specific behaviors.

Optimization	Relational Analog	MongoDB-Specific Aspect
Predicate pushdown (\$match early)	Standard in all CBO systems	No MongoDB-specific aspect
Count without document fetch (T1)	Index-only scan in relational systems	Implementation differs: MongoDB 6.0.3 uses COUNT_SCAN; 8.2.5 uses SBE GROUP
Project while matching (T2)	Projection pushdown in CBO	No MongoDB-specific aspect
Hash aggregation for \$group (T3, T8)	Hash aggregation in relational systems	FFTP loses productivity signal (Advanced = 0)—no CBO analog for this failure mode
Compound index with equality then sort (T6)	Composite index in relational systems	ESR rule is MongoDB-specific; relational systems use cost estimation to choose field order
Early termination via \$limit (T4, T6, T9)	Top-N optimization in relational systems	In MongoDB, early termination is gated by cursor/non-cursor layer split—operators outside \$cursor cannot participate
IXSCAN preference at low selectivity (T7)	CBO avoids this via cardinality estimates	Structural artifact of Advanced/Works metric—no CBO analog; COLLSCAN is systematically underrepresented as a candidate
\$unwind document multiplication (T5)	No relational analog	Specific to document-oriented model; array unnesting multiplies the document stream before \$group, outside the scope of the FFTP mechanism
SBE engine/Works unavailability (T3, T8)	No relational analog	Specific to MongoDB 8.x virtual machine architecture; Works metric is unavailable, forcing FFTP to rely on Score alone

In Table 15, the following abbreviations are used: CBO—Cost-Based Optimization (classical relational optimizer approach based on cardinality estimation and cost modeling);

SBE—Slot-Based Execution engine; ESR—Equality–Sort–Range (recommended field ordering rule for compound indexes); FFTP—First Past the Post (trial-based plan selection mechanism).

The table reveals that most of the observed behaviors have relational analogs at the logical level, while the failure modes that constitute the core empirical contribution of this paper—IXSCAN preference bias, Advanced = 0 degeneration, and \$unwind document multiplication—are specific to the combination of MongoDB’s document-oriented data model and FFTP architecture. In relational systems, these failure modes are either prevented by cost-based cardinality estimation or do not arise at all due to the absence of nested array structures. The ESR compound index principle occupies an intermediate position: while composite indexes are standard in relational systems, the specific field ordering rule and its interaction with index-provided sort elimination are particular to MongoDB’s cursor layer architecture and cannot be directly derived from relational index design principles.

3.7. Effect of Data Distribution on FFTP Behavior

To verify whether the results hold under more realistic conditions, the experiments are repeated on a collection with the same schema and size (50,000 documents) but with a skewed distribution: Electronics accounts for 65% of categories, the US for 55% of regions, shipped and delivered together for 75% of order statuses, prices follow a Pareto distribution, and 70% of orders fall within the last six months of the date range.

The results are organized into three groups. The first group comprises the stable tests—Top-N sort, the ESR index query, grouping by customer_id, and the early termination verification query. All four exhibit identical FFTP behavior, the same winning index, and execution time differences of 0 to 8%. Index-based optimizations are unaffected by data distribution, a finding consistent with the size independence observed in Section 3.4.

The second group comprises tests where the volume of the result stream varies with the distribution. For the \$match + \$group query on category, the number of documents scanned increases from 16,587 to 32,679 because Electronics now dominates the data, and execution time rises from 44 ms to 71 ms. For the \$unwind + \$group query, the number of documents returned increased from 24,836 to 40,170 because orders are concentrated in the last six months of the date range, pushing execution time from 125 ms to 308 ms. In both cases, FFTP behavior and the winning index remain unchanged—what varies is the volume of data being processed, not the plan selection outcome.

The third group contains a single test—the IXSCAN preference bias query—where the effect operates in the opposite direction. Execution time drops from 137 ms to 39 ms under the skewed distribution. Under uniform data, the predicates on discount and total_amount touch approximately 49,000 out of 50,000 documents. Under skewed data, the Pareto-distributed prices produce higher effective selectivity, and the number of documents scanned falls from 48,962 to 14,831. The IXSCAN preference bias itself does not disappear—MongoDB selects idx_total_amount in both cases—but its practical impact is substantially smaller when the data is skewed.

Taken together, these results confirm that the plan selection and FFTP behavior described in Section 3.3 remain qualitatively valid under non-uniform data. The absolute execution times, however, are strongly dependent on data distribution for any query that does not benefit from early termination.

3.8. Limitations

Several limitations should be considered when interpreting the results. First, while Section 3.7 demonstrates that FFTP plan selection is qualitatively stable under skewed data, the absolute execution times are strongly dependent on data distribution for queries that do not benefit from early termination—measurements obtained on uniform data do

not transfer directly to production environments where certain values dominate. Second, the study examines nine pipeline patterns on a single collection with a fixed schema—whether the findings generalize to schemas with different cardinalities, nested documents, or operators such as \$lookup, \$facet, and window functions remains to be verified. Third, all measurements on the local machine are conducted under isolated single-threaded execution—under concurrent workloads, plan cache decisions may differ substantially. The deployment environment comparison in Section 3.5.2 confirms that the behavioral patterns identified on the local machine hold on MongoDB Atlas, but the Atlas cluster runs MongoDB 8.0.20 rather than 8.2.5. Thus, a systematic version-controlled cloud comparison remains a direction for future work.

4. Conclusions

This study examines two distinct but interacting optimization mechanisms in MongoDB aggregation pipelines—FPTP plan selection and Pipeline Optimizer rewriting—and finds that they are orthogonal and complementary. FPTP decides which physical access path to use, while the Pipeline Optimizer rewrites the pipeline structure deterministically based on static rules, with no dependence on runtime data statistics.

FPTP activates in five of the nine tests and selects the optimal plan in four of those cases. The IXSCAN preference bias documented by Tao et al. [4] is confirmed: at an Index Filter Ratio of 0.0001—a value practically equivalent to a full collection scan—MongoDB still selects the index. For pipelines containing blocking operators such as \$group and in-memory \$sort, the SBE engine produces Advanced = 0 for all candidate plans during the trial period, forcing FPTP to differentiate between plans solely on Score. When scores are tied, plan selection becomes non-deterministic. Index-provided sorts emerge as the dominant plan selection signal: a Score of roughly 3.0 against 1.0001 for the alternatives represents an unconditional win that eliminates the physical SORT node entirely. The ESR principle is confirmed to be both version- and size-independent: idx_cat_status_amount consistently achieves 1–2 ms across all three collection sizes in both versions.

The multi-scale analysis at 50K, 150K, and 250K documents identifies three distinct behavioral categories. Queries with index-provided sort and early termination exhibit constant execution time regardless of collection size—a direct consequence of ESR index design and termination after a fixed number of results. Queries without this optimization scale approximately linearly with collection size, while \$unwind + \$group queries exhibit mildly superlinear growth due to document stream multiplication. MongoDB 8.2.5 consistently outperforms 6.0.3 on grouping queries, with execution time reductions of 16% to 28% across all three collection sizes, likely reflecting SBE engine improvements between versions. The sole exception is the unbounded \$match+\$project query, which regresses in 8.2.5 at every tested collection size (+23% at 50K, +3% at 150K, and +14% at 250K). The regression is reproducible and cannot be attributed to a change in execution plan—the number of documents scanned is practically identical between versions—leaving the root cause unverified without server-level profiling.

Taken together, these findings are interpreted within the conceptual framework introduced in Section 3.1. The central hypothesis—that FPTP and the Pipeline Optimizer are orthogonal mechanisms whose interaction can produce suboptimal outcomes that neither would generate in isolation—is fully supported by the experimental evidence. Of the five behavioral dimensions probed by the experimental design, three yield results consistent with theoretical expectations: index-provided sort with early termination produces constant execution time regardless of collection size, ESR compound index design is both version- and size-independent, and FPTP correctly differentiates between candidate plans when the productivity signal is unambiguous. Two dimensions reveal structural limits of the current

optimizer architecture: when blocking operators render `Advanced = 0` for all candidate plans, FFTP degenerates to Score-based tie-breaking that is effectively non-deterministic, and at low predicate selectivity, the IXSCAN preference bias documented by Tao et al. [4] persists unchanged in MongoDB 8.2.5, confirming that this is a structural limitation of the FFTP mechanism rather than a version-specific artifact. The Pipeline Optimizer, by contrast, behaves identically across both versions in all nine tests, confirming that its rule-based, statistics-independent nature renders it robust to version changes but fundamentally unable to compensate for runtime data characteristics that FFTP itself cannot correctly evaluate.

The findings of this study complement and extend two principal lines of prior work. First, with respect to cost-based optimization (CBO), the results confirm the observation of Leis et al. [3] that cardinality estimation errors are the primary source of suboptimal plan selection—but from a different architectural angle. MongoDB’s FFTP mechanism was designed precisely to avoid reliance on cardinality estimates by measuring actual runtime behavior instead. However, the three failure modes identified here show that FFTP introduces its own systematic biases: IXSCAN preference at low selectivity, score degeneration when blocking operators render `Advanced = 0`, and inability to account for `$unwind` document multiplication. These are not estimation errors in the CBO sense—they are structural artifacts of the trial-based selection model itself, and they persist regardless of how accurate the underlying data statistics are. Second, with respect to learned query optimization, the results suggest that the failure modes identified here represent concrete targets for ML-based intervention. The approach of Tao et al. [4], which demonstrated IXSCAN preference bias in MongoDB 7.0.1 for simple `find()` queries, is extended here to multi-stage aggregation pipelines across two major versions and three collection scales, confirming that the bias is both persistent and unaddressed in MongoDB 8.2.5. The Bao optimizer [16] and Kepler [18] demonstrate that learned models can steer plan selection without replacing the underlying optimizer—a strategy directly applicable to the failure modes documented here, where the optimizer generates the correct candidate plans but selects among them suboptimally.

From a practical standpoint, the three identified optimizer limitations translate into concrete index and query design guidelines. IXSCAN preference bias at low selectivity is the most consequential: when a query’s predicates cover a large fraction of the collection, the optimizer may select an index scan over a collection scan regardless of the actual cost, and the only reliable mitigation is to avoid creating indexes on fields with low discriminating power or to restructure the query predicates. Unbounded `$unwind` operations should always be preceded by a selective `$match` and, where possible, followed by an early `$limit` to cap the document stream prior to aggregation. For pipelines containing `$group`, the optimizer has no visibility into output cardinality—the ESR index design principle addresses the pre-group filtering and sorting cost, but post-group ordering remains outside the scope of FFTP. Developers relying on `$group` for Top-N results should place `$sort` and `$limit` explicitly after `$group` and should not expect the optimizer to reorder them automatically.

5. Future Work

The findings of this study are grounded in the architectural properties of MongoDB’s FFTP mechanism rather than in the specific characteristics of the synthetic dataset used. This distinction is important: the three identified failure modes arise from structural properties of the optimizer itself, not from properties of the data. IXSCAN preference bias occurs whenever predicate selectivity is sufficiently low that an index scan covers nearly the entire collection—a condition that can arise in any schema. `Advanced = 0` degeneration occurs whenever a blocking operator such as `$group` appears in the pipeline—regardless of collec-

tion schema or data distribution. Document multiplication after \$unwind occurs whenever a multi-valued array field is unnested before aggregation—a pattern common in real-world document-oriented designs. Consequently, these failure modes are expected to manifest in production environments wherever the triggering conditions are met. What does not transfer directly from the synthetic dataset to real-world deployments are the absolute execution times, which are strongly dependent on data distribution, field cardinality, and document structure.

Four directions for future work are identified based on the limitations and open problems documented in this study. First, the \$match + \$project regression warrants investigation using the MongoDB server profiler, comparing execution statistics between versions to isolate the specific code path responsible for the slowdown. Second, the analysis should be extended to \$lookup, \$facet, and window functions, which are not covered in this study and where the interaction between FFTP and the Pipeline Optimizer is insufficiently understood. Third, FFTP behavior under concurrent workloads warrants dedicated investigation—this study executes all tests in isolation on a single thread, while production environments handle parallel queries that may significantly affect plan cache decisions and trial period outcomes. Fourth, validating the findings on real-world datasets—such as publicly available e-commerce or IoT collections with natural skew, sparse fields, and nested structures—would strengthen the external validity of the identified failure modes and provide a more direct bridge to production deployment scenarios.

Supplementary Materials: The following supporting information can be downloaded at: <https://www.mdpi.com/article/10.3390/info17050488/s1>, Code folder: contains the Python code used to generate all results (fftp_analyzer.py, benchmark.py, and plot_scaling.py); DumpFiles folder: contains dump files generated by the software; Table S1: MongoDB aggregation operators; Figure S1: FFTP plan race for Test 9; Figure S2: Behavior of the MongoDB Pipeline Optimizer for Test 9. Figure S3: Median execution time (ms) by test, version, and collection size. The plot is generated by the plot_scaling.py script based on data in the benchmark dump files (50K collection size, uniform data distribution).

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data supporting the reported results were obtained using software available in the Supplementary Material.

Conflicts of Interest: The author declares no conflicts of interest.

References

1. Harrison, G.; Harrison, M. *MongoDB Performance Tuning: Optimizing MongoDB Databases and Their Applications*; Apress: Berkeley, CA, USA, 2021; ISBN 9781484268797.
2. Chen, T.; Chen, H.; Gao, J.; Tu, Y. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. *Proc. VLDB Endow.* **2023**, *16*, 1777–1789. [\[CrossRef\]](#)
3. Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P.; Kemper, A.; Neumann, T. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* **2016**, *9*, 204–215. [\[CrossRef\]](#)
4. Tao, D.; Liu, E.; Randeni Kadupitige, S.; Cahill, M.; Fekete, A.; Röhm, U. First Past the Post: Evaluating Query Optimization in MongoDB. In *Databases Theory and Applications. ADC 2024; Lecture Notes in Computer Science*; Springer: Singapore, 2025; Volume 15449, pp. 99–113.
5. Rathore, M.; Bagui, S.S. MongoDB: Meeting the Dynamic Needs of Modern Applications. *Encyclopedia* **2024**, *4*, 1433–1453. [\[CrossRef\]](#)
6. Nuriev, M.; Zaripova, R.; Yanova, O.; Koshkina, I.; Chupaev, A. Enhancing MongoDB Query Performance through Index Optimization. In *Proceedings of the E3S Web of Conferences, Yogyakarta, Indonesia, 7–8 August 2024; Volume 531*.

7. Heinrich, R.; Li, X.; Luthra, M.; Kaoudi, Z. Learned Cost Models for Query Optimization: From Batch to Streaming Systems. *Proc. VLDB Endow.* **2025**, *18*, 5482–5487. [[CrossRef](#)]
8. Panwar, V. AI-Driven Query Optimization: Revolutionizing Database Performance and Efficiency. *Int. J. Comput. Trends Technol.* **2024**, *72*, 18–26. [[CrossRef](#)]
9. Karri, N.; Muntala, P.S.R.P. Query Optimization Using Machine Learning. *Int. J. Emerg. Trends Comput. Sci. Inf. Technol.* **2023**, *4*, 109–117. [[CrossRef](#)]
10. Ganesh Chandra, D. BASE Analysis of NoSQL Database. *Future Gener. Comput. Syst.* **2015**, *52*, 13–21. [[CrossRef](#)]
11. Karras, A.; Karras, C.; Samoladas, D.; Giotopoulos, K.C.; Sioutas, S. Query Optimization in NoSQL Databases Using an Enhanced Localized R-Tree Index. In *Information Integration and Web Intelligence. iiWAS 2022; Lecture Notes in Computer Science*; Springer: Cham, Switzerland, 2022; Volume 13635, pp. 391–398.
12. Mouhiha, M.; Mabrouk, A. NoSQL Data Warehouse Optimizing Models: A Comparative Study of Column-Oriented Approaches. *Big Data Res.* **2025**, *40*, 100523. [[CrossRef](#)]
13. Chava, K.; Challa, S.R.; Sriram, H.K.; Chakilam, C.; Kannan, S.; Annapareddy, V.N. Utilizing Query Performance in NoSQL Databases for Applications Based on Big Data. In Proceedings of the 2nd IEEE International Conference on New Frontiers in Communication, Automation, Management and Security (ICCAMS), Bangalore, India, 11–12 July 2025; pp. 1–6.
14. Seethala, S.C. ML-Enhanced SQL and NoSQL Query Optimization for High-Volume Big Data. *Int. J. Sci. Res. Eng. Trends* **2025**, *11*, 1–7.
15. Marcus, R.; Negi, P.; Mao, H.; Zhang, C.; Alizadeh, M.; Kraska, T.; Papaemmanouil, O.; Tatbul, N. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* **2018**, *12*, 1705–1718. [[CrossRef](#)]
16. Marcus, R.; Negi, P.; Mao, H.; Tatbul, N.; Alizadeh, M.; Kraska, T. Bao: Making Learned Query Optimization Practical. In Proceedings of the SIGMOD Record, Philadelphia, PA, USA, 12–17 June 2022; Volume 51, pp. 6–13.
17. Anneser, C.; Tatbul, N.; Cohen, D.; Xu, Z.; Pandian, P.; Laptev, N.; Marcus, R. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* **2023**, *16*, 3515–3527. [[CrossRef](#)]
18. Doshi, L.; Zhuang, V.; Jain, G.; Marcus, R.; Huang, H.; Altinbüken, D.; Brevdo, E.; Fraser, C. Kepler: Robust Learning for Parametric Query Optimization. *Proc. ACM Manag. Data* **2023**, *1*, 109. [[CrossRef](#)]
19. Chen, X.; Chen, H.; Liang, Z.; Liu, S.; Su, H.; Zheng, K.; Wang, J.; Zeng, K. LEON: A New Framework for ML-Aided Query Optimization. *Proc. VLDB Endow.* **2023**, *16*, 2261–2273. [[CrossRef](#)]
20. Sulimov, P.; Lehmann, C.; Stockinger, K. GenJoin: Conditional Generative Plan-to-Plan Query Optimizer That Learns from Subplan Hints. *Proc. ACM Manag. Data* **2025**, *3*, 247. [[CrossRef](#)]
21. Tao, J.; Maus, N.; Jones, H.; Zeng, Y.; Gardner, J.R.; Marcus, R. Learned Offline Query Planning via Bayesian Optimization. *Proc. ACM Manag. Data* **2025**, *3*, 179. [[CrossRef](#)] [[PubMed](#)]
22. Marcus, R. Learned Query Superoptimization (Extended Abstract). In Proceedings of the CEUR Workshop Proceedings, Paris, France, 6–8 December 2023; Volume 3462.
23. Van De Water, R.; Ventura, F.; Kaoudi, Z.; Quiane-Ruiz, J.A.; Markl, V. Farming Your ML-Based Query Optimizer’s Food. In Proceedings of the International Conference on Data Engineering, Online, 9–12 May 2022; Volume 2022, pp. 3186–3189.
24. Negi, P.; Marcus, R.; Wu, Z.; Madden, S.; Kipf, A.; Kraska, T.; Tatbul, N.; Alizadeh, M. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* **2023**, *16*, 1520–1533. [[CrossRef](#)]
25. Hettiarachchi, N.; Yapa, P. OptimAIzerSQL: Optimizing SQL Queries with Heuristic and ML-Based Multi-Agent Systems. In Proceedings of the 2025 5th International Conference on Machine Learning and Intelligent Systems Engineering, MLISE 2025, Shenzhen, China, 13–15 June 2025; pp. 40–48.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.