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

Rosen Ivanov *

Posted Date: 31 March 2026

doi: 10.20944/preprints202603.2514.v1

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

 doi:10.20944/preprints202603.2514.v1

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. It uses MongoDB versions 6.0.3 and 8.2.5 under identical conditions. For each query, it evaluates all valid operator orderings. It also examines the physical execution plan and the Pipeline Optimizer output. Each test runs 20 times. The system clears the plan cache 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 9 times slower than the optimal (80ms vs. 699ms 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 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

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 starts to matter a great deal. Aggregation pipelines with multiple stages are a particular concern: 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 difficult problem 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, 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.

 doi:10.20944/preprints202603.2514.v1

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 different approach altogether. 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 gets 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 wins, gets executed to completion, and is 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 downside is that the trial period covers only a small fraction of full execution, which can introduce systematic biases in plan selection.

MongoDB's aggregation pipeline goes well beyond what a simple find() query can express. The most used aggregation operators and their purposes are described in Supplementary Materials/Table S1. Internally, the pipeline splits into two layers. The first is the cursor stage, where operators like $match, $sort, and $limit can take advantage of indexes and participate in the FPTP race. The second layer contains the remaining pipeline stages — $group, $unwind, $lookup, $project — which run after the cursor stage and sit entirely outside the FPTP mechanism. This split directly shapes how the rule-based Pipeline Optimizer rewrites operator sequences, and it determines when the optimizer is likely to make a poor choice. 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 meaningful throughput gains for pipelines with $group or $lookup through batch processing. The trade-off 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 FPTP — have catalyzed an extensive research programme on learned query optimization, in which machine learning models are employed 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 substituting 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 a focus on the FPTP 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. 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 relative to 6.0.3.

 doi:10.20944/preprints202603.2514.v1

The objective of this study is to empirically characterize the behavior of MongoDB's FPTP 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 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 NoSQL systems broadly. The closest prior work to this paper is the study by Tao et al. [4], which is the first systematic experimental look at MongoDB's FPTP optimizer. Published at the Australasian Database Conference (ADC 2024) and included in Lecture Notes in Computer Science (Springer), the paper shows that FPTP 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 trace this to two concrete problems: collection scans are frequently not generated as candidates at all, and the Advanced/Works metric misjudges the true cost of index access because it does not account for the extra step of fetching the actual document after finding 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 this paper follows. 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 hands-on angle — their treatment of MongoDB performance tuning covers everything from schema design to aggregation pipeline construction and index selection and puts the optimization problem 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.

Ten years ago, applying machine learning to query optimization was a niche idea. Today it is one of the most active research directions in the database community. The shift started with a simple but uncomfortable observation: traditional optimizers break down because their cardinality estimates are wrong, often by orders of magnitude — a point Leis et al. [3] made rigorously and with lasting effect. The early systems tried to do everything 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

 doi:10.20944/preprints202603.2514.v1

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, trained a model to predict the winning plan directly from parameter values, and fall back to the default optimizer when the model is uncertain.

Later 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 run thousands of times, Tao et al. [21] and Marcus [22] argued that spending more time finding a near-optimal plan once is worth it. Van De Water et al. [23] addressed a problem that often goes unspoken — where does the training data come from? Negi et al. [24] asked what happens when the workload drifts and 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 focused on relational SQL systems with cost-based optimizers, the present paper addresses a fundamentally different setting: the FPTP optimizer of MongoDB operating on aggregation pipelines in a document-oriented NoSQL context. No prior work, to the best of our knowledge, systematically characterizes the failure modes of the FPTP mechanism for multi-stage aggregation queries, nor empirically quantifies the performance impact of operator ordering across versions and collection scales.

3. Experiments

3.1. Database Description

3.1.1. Document Structure

The experiments use a synthetic orders collection in a database named fptp_analysis_db, running on MongoDB 6.0.3 and 8.2.5 on a local server. The collection holds 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.

Table 1. Document structure of the orders collection.

Field	Type	Description / Values
order_id	String	Unique identifier: ORD-0000001 … ORD-0050000
customer_id	String	5,000 distinct customers: CUST-00001 … CUST-05000
product	String	Laptop, Phone, Tablet, Monitor, Keyboard
category	String	Electronics, Clothing, Food, Sports, Books
status	String	shipped, delivered, cancelled, 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	total_amount = quantity × unit_price × (1 − discount)
order_date	Date	2022-01-01 – 2023-12-31
tags	Array	0–3 elements drawn from a fixed tag vocabulary

3.1.2. Index Configuration

 doi:10.20944/preprints202603.2514.v1

Eight indexes are defined to cover 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 }	ingle-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

The ESR (Equality - Sort - Range) rule defines 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.

3.2. Metrics

Three metrics are recorded for each candidate plan during the FPTP trial period.

Advanced is 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 productive.

Works is the total number of work units performed by the plan. By definition, Works ≥ Advanced always. The difference between the two represents needTime steps — for example, scanning index keys that do not match the query predicate.

Score is an internal FPTP metric. In the classic engine, Score ≈ Advanced / Works + 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 FPTP has no productivity signal to work with and must differentiate between plans using Score alone.

3.3. Experimental Results

All the results in this article were collected using two custom-built Python scripts. The fptp_analyzer.py script (Supplementary Materials/Code/fptp_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") runs all candidate plans in parallel during the trial period and returns the FPTP 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

 doi:10.20944/preprints202603.2514.v1

Materials/Code/benchmark.py). All raw results are written to JSON dump files for offline analysis (Supplementary Materials/DumpFiles/ *.json).

All tests are run 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 only 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 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, FPTP race results for both versions, and a comparative discussion are provided.

The fptr_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 uses a straightforward pipeline that filters orders by status and counts the results. The query is shown in Listing 1. The goal is to check whether FPTP 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'  }
]
```

In both versions, FPTP does not engage — there are no competing plans to race. The more interesting finding is the difference in 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, 4ms. In MongoDB 8.2.5 the winning stage is GROUP, executed by the SBE engine, also at 4ms. This is not 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 applied and there are no competing plans, FPTP has nothing to compete with and is not activated.

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

This test examines FPTP behavior when a compound index (idx_status_region) competes against two single-field indexes. The hypothesis is that the compound index will dominate, since it covers both predicates in a single scan. 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}  }
]
```

 doi:10.20944/preprints202603.2514.v1

The results are summarized in Table 3.

Table 3. Test 2 — FPTP race results (3 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

FPTP activates in both versions with three competing plans, and the winning strategy is identical in both — execution time is 6.5ms in 6.0.3 and 8ms in 8.2.5, a 23% slowdown in the newer version. Plan 0 achieves Advanced = Works = 101, meaning every index key scan 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 around 19–25% of fetched documents passing the filter — hence the low Advanced counts. FPTP correctly identifies the compound index as the dominant plan.

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

This test looks at 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}     }
]
```

The results are summarized in Table 4.

Table 4. Test 3 — FPTP race results (2 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

Test 3 exposes a structural problem 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 chosen by its position in the plan array, not because it outperforms the alternative on any metric. The key difference between versions is in the availability of Works: in 6.0.3, Works=0 because the SBE engine

 doi:10.20944/preprints202603.2514.v1

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 42ms in 6.0.3 to 31ms in 8.2.5, a 26% gain. Despite idx_cat_status_amount being the theoretically better choice — it covers both category and status — FPTP cannot identify it as such when the scores are tied. The problem is structural: resolving it would require statistics on $group output cardinality, which FPTP does not have access to.

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

This test examines FPTP behavior on a Top-N query. The hypothesis is that idx_total_amount, being a descending index, provides index-provided sort and allows early termination after 100 documents. 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}  }
]
```

The results are summarized in Table 5.

Table 5. Test 4 — FPTP race results (3 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

Plan 0 wins outright in both versions. idx_total_amount is a descending index — MongoDB scans 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 shortcut: 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 is a decisive margin. Execution time is 2ms in both versions. The Pipeline Optimizer applies the same transformation in both versions, collapsing 4 logical stages into a single $cursor with index-provided sort (*).

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

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

 doi:10.20944/preprints202603.2514.v1

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  }
]
```

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
FPTP 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
FPTP active	no (0 plans)	no (0 plans)	none

FPTP does not activate in either version — only one index applies to the date range predicate, leaving nothing to race. After $match, 25,045 documents pass the filter; after $unwind, the document count grows to approximately 38,000, at an average of around 1.5 tags per document; the final output is 20 documents. Winning Plan Stage is PROJECTION_SIMPLE. $cursor absorbs $match + $project. The main 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 order is undefined relative to any index.

Test 6: Optimal compound index (ESR rule)

This test is an empirical verification of the Equality → Sort → Range (ESR) rule. The rule defines 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 through 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}  }
]
```

The results are summarized in Table 7.

 doi:10.20944/preprints202603.2514.v1

Table 7. Test 6 — FPTP race results (4 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 stable result in the study: 1ms 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 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 index design is confirmed as version independent.

Test 7: IXSCAN preference bias

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

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

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

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 chooses the suboptimal plan

FPTP does not activate in either version — there are no competing plans. The interesting finding here is how the two versions differ in their plan choice. MongoDB 6.0.3 picks COLLSCAN at 163ms — 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 makes IXSCAN more expensive than

 doi:10.20944/preprints202603.2514.v1

scanning the collection outright. MongoDB 8.2.5 picks IXSCAN[idx_total_amount] at the same selectivity and runs in 134ms — not because the choice is better, but because the specific combination of data 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 is still present in 8.2.5 — and under worse conditions than before. Where 6.0.3 correctly fell back to COLLSCAN at ratio=0.0001, 8.2.5 chooses the index regardless. The bias has not been fixed.

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

This test analyzes cancelled 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. Cancelled 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}        },
```

The results are summarized in Table 9.

Table 9. Test 8 — FPTP race results (3 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 chosen by position, FPTP correctly differentiates the plans here. idx_status_region covers both $match predicates and scans only 3,166 documents, giving an Index Filter Ratio of 0.7429. This translates into a clear Score advantage: 2.0002 for Plan 0 against roughly 1.19 for the alternatives. As in Test 3, Works=0 in 6.0.3 due to the SBE engine not tracking work units, while Works=101 is available in 8.2.5. Execution time is 16.5ms in 6.0.3 and 15ms in 8.2.5, a 9% improvement.

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

This test verifies the early termination effect when $limit is placed at position 3 out of 4 stages in the pipeline. The query is shown in Listing 9.

 doi:10.20944/preprints202603.2514.v1

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}
        }
```

The results are summarized in Table 10.

Table 10. Test 9 — FPTP race results (3 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 shows FPTP behaving consistently 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 is satisfied. Plans 1 and 2 require a physical sort node and cannot return a single 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 seen in Tests 4 and 6. The small numerical differences between versions — docsExamined=25 in 6.0.3 versus 36 in 8.2.5, Score=2.4002 versus 2.2780 — fall within the expected range given different random data distributions. The fact that Works is available in both versions confirms that Works=0 in 6.0.3 is specific to the SBE engine and $group pipelines, not a general limitation of the FPTP mechanism.

3.4. Summary of Results

Table 11 summarizes the 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

 doi:10.20944/preprints202603.2514.v1

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 activity 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 exists. The Pipeline Optimizer applies the same transformations in both versions without exception. Under controlled conditions — identical hardware, medians over 20 repeated runs — MongoDB 8.2.5 shows improved performance on T3 (−26%), T5 (−21%), and T7 (−18%). The only regression is T2 (+23%), discussed in Section 3.3. Queries backed by an ESR-ordered index are version-independent: T4, T6, and T9 hold steady at 1–2ms in both versions.

3.4.1. Execution Time Analysis

All measurements are taken 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, 3.20GHz base clock, 4.50GHz Turbo), with 16GB DDR4 3200MHz and a KIOXIA KBG40ZNS512G 512GB 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 side by side in separate directories, each with its own data path, log path, and network port to prevent any resource conflicts. The two instances never run at the same time: each version is started, tested, and shut down before the other is launched.

Tests are run at three collection sizes — 50K, 150K, and 250K documents — using the same schema and index configuration throughout. Each version-and-size combination runs 20 times, with 2 warmup runs and a planCacheClear before each measurement. Median values are used as the primary metric because of their robustness to outliers, particularly for sub-5ms 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).

 doi:10.20944/preprints202603.2514.v1

The results fall into three groups based on how performance scales with collection size. T4, T6, and T9 show constant execution time across all three sizes in both versions, staying in the 1–3ms range throughout. The reason is the same 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 the right index design combined with early termination is both version-independent and size-independent.

Tests T1, T2, T5, T7, and T8 scale approximately linearly in both versions. T5 ($unwind + $group) is slightly superlinear: execution time grows by roughly 5.6× in MongoDB 6.0.3 (130.5ms → 735.5ms) and 6.5× in MongoDB 8.2.5 (102.5ms → 668.5ms) for a 5× increase in data volume. The extra growth comes from $unwind multiplying the document stream before it reaches $group, so the effective input to the aggregation grows faster than the collection itself. T7 (IXSCAN preference bias) scales at roughly 4.5× for 6.0.3 (163ms → 740.5ms) and 5.2× for 8.2.5 (133.5ms → 699ms) - 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) shows moderate superlinear scaling in both versions, at roughly 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 on T3 at every collection size — by 26% at 50K, 28% at 150K, and 16% at 250K — which likely reflects improvements to the SBE engine's $group implementation between versions.

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

3.5. MongoDB Pipeline Optimizer

3.5.1. Purpose and Operating Principle

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

The result of the rewriting is visible 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 a split of the pipeline into two layers. The first is the $cursor stage, which contains the physical operators that run directly against the index — IXSCAN or COLLSCAN, FETCH, and optionally SORT, LIMIT, and PROJECTION if they can be satisfied through the index. The second layer holds the remaining pipeline stages that execute after the cursor — $group, $unwind, $lookup, and any $sort or $limit that cannot be pushed into the index scan.

Three optimizations are applied. First, predicate pushdown: $match is moved as early as possible in the pipeline to reduce the number of documents that subsequent stages need to process. Second, index-provided sort: if an available index already 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 execution as possible, allowing MongoDB to stop processing documents as soon as the required count is reached.

3.5.2. Results

The Pipeline Optimizer behaves identically in MongoDB 6.0.3 and 8.2.5 — every transformation is applied in the same way across both versions. Table 13 summarizes the optimizer's behavior for all

 doi:10.20944/preprints202603.2514.v1

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 13. 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 в 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

Index-provided sort, seen 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 4 stages down to 1. It requires an exact match between the $sort field and direction and the corresponding index key. For SBE queries in Tests 3, 7, and 8, $group runs inside the cursor stage, but any $sort that follows $group always stays as a separate stage — $group is a blocking operation, and its output order bears no relationship to any index. Test 5 is the only case where the visible pipeline structure comes out unchanged: $unwind blocks any reordering of the stages that follow it. The fact that the Pipeline Optimizer behaves identically in both versions confirms that its transformations are rule-based and static, independent of runtime data characteristics or which execution engine is in use.

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 it had already decreased compared to earlier versions. The present study finds it still present and unchanged in 8.2.5.

3.6. Effect of Data Distribution on FPTP Behavior

To check 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, 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 split into three groups. The first group covers the stable tests — Top-N sort, the ESR index query, grouping by customer_id, and the early termination verification query. All four show identical FPTP behavior, the same winning index, and execution time differences of 0 to 8%.

 doi:10.20944/preprints202603.2514.v1

Index-based optimizations are unaffected by data distribution, a finding that aligns with the size-independence observed in Section 3.4.

The second group covers tests where the volume of the result stream changes with the distribution. For the $match + $group query on category, the number of documents scanned grows from 16,587 to 32,679 because Electronics now dominates the data, and execution time climbs from 44ms to 71ms. For the $unwind + $group query, the number of documents returned grew from 24,836 to 40,170 because orders are now concentrated in the last six months of the date range, pushing execution time from 125ms to 308ms. In both cases, FPTP behavior and the winning index stay the same — what changes is the volume of data being processed, not how the plan is selected.

The third group contains just one test — the IXSCAN preference bias query — where the effect goes in the opposite direction. Execution time drops from 137ms to 39ms under the skewed distribution. Under uniform data, the predicates on discount and total_amount touch close to 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 go away — MongoDB picks idx_total_amount in both cases — but its practical impact is much smaller when the data is skewed.

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

3.7. Limitations

Several limitations should be kept in mind when interpreting the results. First, while Section 3.6 shows that FPTP plan selection is qualitatively stable under skewed data, the absolute execution times depend heavily on data distribution for queries that do not benefit from early termination — numbers measured on uniform data do not transfer directly to production environments where certain values dominate. Second, the study covers 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 has not been verified. Third, all measurements are taken under isolated single-threaded execution on one physical machine — under concurrent workloads or on different hardware, both the absolute execution times and plan cache decisions may differ.

4. Conclusions

This study examines two 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 5 of the 9 tests and selects the optimal plan in 4 of them. 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 chooses the index. For pipelines with 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 solely on Score. When scores are tied, the choice is non-deterministic. Index-provided sorts stand out as the dominant plan selection signal: a Score of roughly 3.0 against 1.0001 for the alternatives is an unconditional win that eliminates the physical SORT node entirely. The ESR principle is confirmed as both version- and size-independent: idx_cat_status_amount consistently achieves 1–2ms across all three collection sizes in both versions.

The multi-scale analysis at 50K, 150K, and 250K documents reveals three behavioral categories. Queries with index-provided sort and early termination show constant execution time regardless of collection size — a direct consequence of the ESR index design and stopping after a fixed number of

 doi:10.20944/preprints202603.2514.v1

results. Queries without this optimization scale approximately linearly, while $unwind+$group queries show mildly superlinear growth due to document stream multiplication. MongoDB 8.2.5 consistently outperforms 6.0.3 on grouping queries, with reductions of 16% to 28% across all three collection sizes, likely reflecting SBE engine improvements between versions. The one exception is the unbounded $match+$project query, which regresses in 8.2.5 at every tested collection size (+23% at 50K, +3% at 150K, +14% at 250K). The regression is reproducible and cannot be explained by a change in execution plan — the number of documents scanned is practically identical between versions — leaving the root cause unverified without server-level profiling.

From a practical standpoint, the three identified optimizer limitations translate into concrete 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 choose 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 before 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 the post-group ordering remains outside the scope of FPTP. Developers relying on $group for Top-N results should place $sort and $limit explicitly after $group and not expect the optimizer to reorder them automatically.

Three directions for future work are identified. First, the $match+$project regression should be investigated using the MongoDB server profiler, comparing execution statistics between versions to isolate the specific code path responsible. Second, the analysis should be extended to $lookup, $facet, and window functions, which are not covered in this study and where the interaction between FPTP and the Pipeline Optimizer is less well understood. Third, FPTP behavior under concurrent workloads deserves attention — this study runs all tests in isolation on a single thread, while production environments handle parallel queries that may affect plan cache decisions and trial period outcomes.

Supplementary Materials: The following supporting information can be downloaded at the website of this paper posted on Preprints.org, Code folder: contains the Python code used to generate all results (fptp_analyzer.py, benchmark.py, and plot_scaling.py); DumpFiles folder: contains dump files generated by the software; Table S1: MongoDB aggregation operators; Figure S1: FPTP plan race for Test 9; Figure S2: Behavior of the MongoDB Pipeline Optimizer for Test 9.

Funding: This research received no external funding.

Informed Consent Statement: Not applicable.

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

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

References

1. Harrison, G.; Harrison, M. MongoDB Performance Tuning: Optimizing MongoDB Databases and Their Applications; 2021; ISBN 9781484268797.
2. Chen, T.; Chen, H.; Gao, J.; Tu, Y. LOGER: A Learned Optimizer towards Generating Efficient and Robust Query Execution Plans. In Proceedings of the Proceedings of the VLDB Endowment; 2023; Vol. 16, pp. 1777–1789.
3. Leis, V.; Gubichev, A.; Mirchev, A.; Boncz, P.; Kemper, A.; Neumann, T. How Good Are Query Optimizers, Really? In *Proceedings of the VLDB Endowment*; 2016; Vol. 9, pp. 204–215.
4. Tao, D.; Liu, E.; Randeni Kadupitige, S.; Cahill, M.; Fekete, A.; Röhm, U. First Past the Post: Evaluating Query Optimization in MongoDB. In Proceedings of the Lecture Notes in Computer Science (including

 doi:10.20944/preprints202603.2514.v1

subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); 2025; Vol. 15449 LNCS, pp. 99–113.

5. Rathore, M.; Bagui, S.S. MongoDB: Meeting the Dynamic Needs of Modern Applications. *Encyclopedia* **2024**, *4*, 1433–1453, doi:10.3390/encyclopedia4040093.
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; 2024; Vol. 531.
7. Heinrich, R.; Li, X.; Luthra, M.; Kaoudi, Z. Learned Cost Models for Query Optimization: From Batch to Streaming Systems. In Proceedings of the Proceedings of the VLDB Endowment; 2025; Vol. 18, pp. 5482–5487.
8. Panwar, V. AI-Driven Query Optimization: Revolutionizing Database Performance and Efficiency. *International Journal of Computer Trends and Technology* **2024**, *72*, 18–26, doi:10.14445/22312803/ijctt-v72i3p103.
9. Karri, N., Muntala, P. S. R. P. Query Optimization Using Machine Learning. *International Journal of Emerging Trends in Computer Science and Information Technology* **2023**, *4*, 109-117.
10. Ganesh Chandra, D. BASE Analysis of NoSQL Database. *Future Generation Computer Systems* **2015**, *52*, 13–21, doi:10.1016/j.future.2015.05.003.
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 Proceedings of the Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); 2022; Vol. 13635 LNCS, pp. 391–398.
12. Mouhiha, M.; Mabrouk, A. NoSQL Data Warehouse Optimizing Models: A Comparative Study of Column-Oriented Approaches. *Big Data Research* **2025**, *40*, doi:10.1016/j.bdr.2025.100523.
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), 2025, pp. 1–6.
14. Seethala, S. C. ML-Enhanced SQL and NoSQL Query Optimization for High-Volume Big Data. In Proceedings of the Financial and Healthcare Applications, 2025.
15. Marcus, R.; Negi, P.; Mao, H.; Zhang, C.; Alizadeh, M.; Kraska, T.; Papaemmanouil, O.; Tatbul, N. Neo: A Learned Query Optimizer. In Proceedings of the Proceedings of the VLDB Endowment; 2018; Vol. 12, pp. 1705–1718.
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; 2022; Vol. 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. In Proceedings of the Proceedings of the VLDB Endowment; 2023; Vol. 16, pp. 3515–3527.
18. Doshi, L.; Zhuang, V.; Jain, G.; Marcus, R.; Huang, H.; Altinbüken, D.; Brevdo, E.; Fraser, C. Kepler: Robust Learning for Parametric Query Optimization. *Proceedings of the ACM on Management of Data* **2023**, *1*, 1–25, doi:10.1145/3588963.
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. In Proceedings of the Proceedings of the VLDB Endowment; 2023; Vol. 16, pp. 2261–2273.
20. Sulimov, P.; Lehmann, C.; Stockinger, K. GenJoin: Conditional Generative Plan-to-Plan Query Optimizer That Learns from Subplan Hints. *Proceedings of the ACM on Management of Data* **2025**, *3*, 1–25, doi:10.1145/3749165.
21. Tao, J.; Maus, N.; Jones, H.; Zeng, Y.; Gardner, J.R.; Marcus, R. Learned Offline Query Planning via Bayesian Optimization. *Proceedings of the ACM on Management of Data* **2025**, *3*, 1–29, doi:10.1145/3725316.
22. Marcus, R. Learned Query Superoptimization (Extended Abstract). In Proceedings of the CEUR Workshop Proceedings; 2023; Vol. 3462.

 doi:10.20944/preprints202603.2514.v1

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 Proceedings - International Conference on Data Engineering; 2022; Vol. 2022-May, 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. In Proceedings of the Proceedings of the VLDB Endowment; 2023; Vol. 16, pp. 1520–1533.
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; 2025; pp. 40–48.