

Performance Degradation After Migrating from Regex to Atlas Search

Date: July 22, 2025

1. Executive Summary (TL;DR)

We recently migrated a query from using MongoDB's native `$regex` operator to Atlas Search (`$search`) with the goal of improving performance and reducing resource consumption.

However, post-deployment monitoring shows the **opposite effect**. The new Atlas Search implementation is significantly **slower** and consumes **more CPU and memory** than the original regex query. This report details the implementation and performance metrics.

2. Background

The targeted feature involves searching for documents (users, giggers, etc.) by fields like `name` or `email`. The original implementation used a case-insensitive `$regex` search. While functional, it was identified as a resource bottleneck, causing high CPU load because it couldn't efficiently use standard indexes, leading to collection scans.

The migration to Atlas Search was intended to leverage its powerful, optimized indexing to provide faster and less resource-intensive search capabilities.

3. Implementation Details

Here is a simplified comparison of the aggregation pipeline stages before and after the change.

Previous Implementation (`$regex`)

This code uses a standard `$match` stage with a `$regex` operator to find partial, case-insensitive matches.

Aggregation Pipeline (`Get All`):

```
1 [{"match":{"country":{"eq":"SG"}}}, {"sort":{"name":1}}, {"skip":0}, {"limit":20}]
```

Aggregation Pipeline (with `name search`):

```
1 [{"match":{"country":{"eq":"SG"}}}, {"addFields":{"mobile":{"toString":"mobile_no"}}}, {"addFields":{"full_mobile":{"concat":["country_code", "mobile"]}, "full_mobile2":{"concat":["country_code", "mobile"]}}}, {"match":{"or":[{"name":{}}, {"email":{}}, {"mobile_no":{}}, {"mobile":{}}, {"full_mobile":{}}, {"full_mobile2":{}}]}}, {"sort":{"name":1}}, {"skip":0}, {"limit":20}]
```

Current Implementation (`$search`)

The new code uses a `$search` stage.

Atlas Search Index Definition (`giggers_full_text_search`):

```
1 {
```

```

2  "mappings": {
3    "dynamic": false,
4    "fields": {
5      "_id": {
6        "type": "objectId"
7      },
8      "country": {
9        "type": "token"
10     },
11     "email": {
12       "analyzer": "lucene.keyword",
13       "type": "string"
14     },
15     "mobile_no": {
16       "analyzer": "lucene.standard",
17       "type": "string"
18     },
19     "name": {
20       "analyzer": "lucene.standard",
21       "type": "string"
22     },
23     "resume_link": {
24       "type": "string"
25     },
26     "status": {
27       "type": "token"
28     },
29     "staffing_agency_id": {
30       "type": "objectId"
31     }
32   }
33 }
34 }
35

```

Aggregation Pipeline (Get All):

```

1  [{"$match":{"showAllDocuments":"true"}},{"$search":{"index":"giggers_full_text_search","compound":
{"filter":[{"equals":{"path":"country","value":"SG"}}]}}, {"$sort":{"name":1}}, {"$facet":{"metadata":
[{"$count":"totalCount"}], "data":[{"$skip":0}, {"$limit":20}]}}]

```

Aggregation Pipeline (with name search):




```

1  [{"$match":{"showAllDocuments":"true"}},{"$search":{"index":"giggers_full_text_search","compound":
{"filter":[{"equals":{"path":"country","value":"SG"}]},"should":[{"compound":{"must":[{"wildcard":
{"query":"Vikash*","path":"name","allowAnalyzedField":true}}]}}, {"compound":{"must":[{"wildcard":
{"query":"Vikash*","path":"email","allowAnalyzedField":true}}]}}, {"compound":{"must":[{"wildcard":
{"query":"Vikash*","path":"mobile_no","allowAnalyzedField":true}}]}]},"minimumShouldMatch":1}}}, {"$sort":
{"name":1}}, {"$facet":{"metadata":[{"$count":"totalCount"}], "data":[{"$skip":0}, {"$limit":20}]}}]

```

4. Performance Comparison

The following table summarizes the performance metrics observed in our production environment before and after the deployment. The results are contrary to our expectations.

Metric	Previous (<code>\$regex</code>)	Current (<code>\$search</code>)	Impact
Avg. Query Time	~700ms	~15s	 ~2042.86% Increase
CPU Utilization	1% (with spikes to 5%)	> 80% (sustained)	 Higher Sustained Load
Memory Usage	Moderate	High	 Significant Increase