CASE STUDY

SQUARESHIFT

Optimizing and Scaling Elasticsearch Cluster for High Performance and Multi-Tenancy for a Leading Telecommunications Provider

About the Client

The client is a communication and markets technology company powered by interconnected platforms: messaging, voice, directory and analytics.

Project Context

The client's on-premises Elasticsearch cluster was running on version 7.17.12, focused on achieving high ingestion speeds, efficient multi-tenancy, effective monitoring, and implementing a sustainable Index Lifecycle Management (ILM) strategy. They wanted SquareShift team to suggest practical recommendations and best practices with scenario-based explanations and step-by-step approaches.

Scope of work and recommendation

1. Monitoring Elasticsearch Cluster

Effective cluster monitoring is essential for maintaining optimal performance. Here are key practices:

- **Use Dedicated Monitoring Cluster**: Offloading monitoring to a separate cluster minimizes the impact on production nodes.
- **Alerts and Thresholds**: Configure alerts for critical metrics such as:
  - CPU usage over 80%
  - Heap usage above 75%
  - Node availability and failure detection

Machine Learning for Anomaly Detection: Use anomaly detection in the Elastic Stack to monitor indexing rates and search latency for abnormal patterns.

2. Improving Ingestion Speed

High ingestion speed is crucial for handling large datasets in real-time. Recommendations include:

- **Bulk Indexing Optimization:**
  - Use bulk requests and adjust batch sizes based on cluster capacity.
  - Set the refresh interval to -1 during ingestion to prevent index refreshes that slow down performance.
- **Indexing Pressure Configuration:**

Adjust indexing_pressure.memory.limit to manage memory usage during high-load indexing.

3. Use of Aliases

Aliases provide flexibility in managing indices, allowing for seamless data management and querying:
Write Aliases: Direct writes to specific indices without service disruption, ideal for zero-downtime migrations.
Query Routing: Route queries to specific sets of indices to improve data access control and simplify multi-tenancy management.

Optimizing and Scaling Elasticsearch Cluster for High Performance and Multi-Tenancy for a Leading Telecommunications Provider

4. Multi-Tenancy Recommendations

For multi-tenant environments, configure the cluster for data isolation and efficient query handling:

- **Tenant-based Indexing:** Assign each tenant its own index or a set of indices, balancing scalability and isolation.
- **Role-Based Access Control (RBAC):** Use field- and document-level security to limit tenant access to their respective data.

5. Index Lifecycle Management (ILM) Strategy

Implement ILM for optimized storage and query efficiency. A structured ILM can reduce storage costs and manage data retention. **Case Study:** Set up hot, warm, and cold phases as follows:

- **Hot Phase**: Active, frequently queried data resides here. Set short retention to reduce load.
- **Warm Phase**: Data is moved to warm nodes for infrequent access.

Cold Phase: Rarely accessed data resides here. Use lower-cost storage if available.

6. Shard Allocation – Scenario-Based Explanation

Proper shard allocation ensures high performance and reliability. Recommendations for specific scenarios include:

- **High-Availability Shard Allocation**: Distribute replicas across availability zones for fault tolerance.

Shard Sizing: Monitor shard sizes (10–50GB recommended) and use _split API for resizing after index creation if necessary.

7. Searching in the Hot Phase Only

To reduce latency in time-sensitive queries, limit searches to hot-phase indices using index patterns or aliases that restrict queries to hot-tier data.

8. Search Latency Comparison Across Phases

Typically, latency increases as data moves from hot to cold phases due to slower storage types in warm/cold tiers. Track latencies across phases in Kibana's monitoring dashboard.

9. Adjusting Shard Sizes Post-Creation

After index creation, shard sizes can be increased with the _split API. This operation requires downtime for the affected indices and should be tested on a staging environment before applying to production.

10. Switching Hardware Profiles

Switching from CPU-optimized to storage-optimized profiles affects storage speed and indexing rate:

- **CPU-Optimized**: Beneficial for indexing-intensive workloads.
- **Storage-Optimized**: Recommended for long-term data storage with less frequent access.

11. Logstash Configuration for Multi-Threading

Configuring Logstash for higher throughput requires setting pipeline.workers and pipeline.output.workers to values suited to system resources, ensuring efficient use of CPU and memory.

Example Logstash Configuration:

```
yaml
Copy code
pipeline:
  workers: 16
  output:
    workers: 8
```

12. Replica Count Update in Index Templates

After setting the replica count in an index template, apply changes to existing indices by updating _settings:

```
PUT /index_name/_settings
{
  "number_of_replicas": 2
}
```

13. Targeting Specific Replicas

While targeting specific replicas isn't supported, shard routing can distribute load across nodes with replicas.

14. Monitoring Cluster Sizing & Retention

A monitoring cluster typically requires nodes with storage-optimized profiles. Retention is configured based on data volume, with a common period of 7-30 days.

15. Field Optimization Based on Mappings

Optimized field mapping is critical for efficient storage and query performance. Avoid indexing unnecessary fields by setting index to false for fields not needed in searches.

16. Data Movement Across Phases

To manage ILM transitions, monitor policies using _ilm/policy/explain. Manually adjust index states if necessary to reflect updated ILM stages.

17. Snapshot Policy for Cloud Backup

To ensure data resilience, increase snapshot frequency based on the ingestion rate. Run snapshots in parallel if high throughput is required.

18. Adjusting Index Buffer Size

Use indexing_pressure.memory.limit for indexing-heavy operations to prevent memory overload during peak indexing.

19. Disabling/Enabling Indices

Close and open indices as needed using the _close and _open APIs.

20. Bulk Delete API and Regex

While bulk delete API doesn't support regex, client-side filtering can batch-delete documents with specific patterns.

21. Bulk Request Limit for Delete API

Test batch sizes under 1000, increasing based on server capacity.

22. Failed Task Retrying
Retry failed tasks with a queue-based approach and exponential backoff to ensure smooth retries.

23. Force Merge vs. Segment Merge
Forcibly merging segments (_forcemerge) is useful for storage optimization after indexing completion.

24. Refresh Interval and Document Count
Setting refresh_interval to -1 prevents updating document counts until a manual refresh, ensuring faster indexing.

25. Translog Durability and Data Integrity
With async durability, integrity is monitored by observing disk-write latency. High disk I/O can lead to potential delays.

26. Impact of index: false
Disabling field indexing reduces storage size but sacrifices searchability. Recommended for fields needed only in response payloads.

27. Delay in Auto-scaling
Delays in scaling can occur with high network latency or cluster resource constraints. Use zone-aware node allocation for improved response time.

28. ILM Rollover Issue
Ensure rollover criteria match actual data patterns. For ILM rollover, size, age, and document count must be aligned in policy configuration.

29. Dynamic Memory Scaling
Elastic Cloud supports temporary scaling of memory. However, frequent changes may impact stability, especially during peak indexing.

30. Hot-to-Warm Phase Transitions
Check _ilm/explain for errors in phase transitions. Adjust ILM policy if indices do not move as expected.

31. Terraform Parameters for Monitoring Retention
Use Terraform configurations for setting monitoring data retention, which helps control cluster storage use.

32. Hardware Profile Impact on Latency
Switching hardware profiles, especially from CPU- to storage-optimized, impacts latency. Storage-optimized profiles improve storage efficiency for large datasets.

33. High CPU Usage During Peak Read Operations
To optimize reads:

- Adjust query patterns and caching
- Optimize mappings, avoiding high-cardinality fields