ADVANCED OPTIMIZATION OF LARGE-SCALE SEARCH FUNCTIONALITY USING ELASTICSEARCH

Sipilov I.
Head of Platform Design at Citadel

ABSTRACT
This article presents a detailed case study of the application of Elasticsearch in optimizing large-scale search functionality for the recruitment platform NannyServices.ca. By integrating Elasticsearch's core algorithms such as inverted indexing, BM25 scoring, and sharding techniques with custom user-driven relevance models, we enhanced both the speed and accuracy of search results. The system efficiently indexed 300,000 profiles within 240 seconds, dramatically improving the search experience for thousands of users. This paper provides a deep dive into the architectural decisions, algorithmic customizations, and performance benchmarks, underlining the mathematical foundation and technological advantages of Elasticsearch in large-scale search applications.

Keywords: Elasticsearch optimization, large-scale search, inverted indexing, BM25 scoring, relevance ranking, distributed search architecture, geospatial filtering, search latency reduction, custom search algorithms, Elasticsearch performance benchmarks, search scalability, Elasticsearch sharding, function score queries, search system architecture, user-driven relevance models, recruitment platform optimization, real-time search queries, indexing strategy.

1. Introduction

- **Background**: NannyServices.ca is a platform connecting families with childcare professionals. Due to the rapidly increasing number of profiles (300,000+), the platform required a more robust search system to deliver relevant results quickly. The need for both speed and relevance was driven by user expectations and the platform's business model, which depended on matching families with qualified nannies in a timely manner.
- **Objective**: This article explores how Elasticsearch was leveraged, with advanced mathematical algorithms and distributed computing, to solve the challenges of real-time, relevant search at scale. The primary goal was to reduce search latency, increase result relevance, and ensure scalability for future growth.

2. Overview of Elasticsearch Architecture and Search Algorithms

- **2.1 Elasticsearch Core Components**
- **Inverted Index**: At the core of Elasticsearch is the inverted index, a data structure that maps terms to documents. Unlike a traditional forward index, where a document points to the terms it contains, the inverted index allows Elasticsearch to quickly retrieve documents based on search terms.
- **Lucene as a Foundation**: Elasticsearch is built on Apache Lucene, a high-performance text search engine library. Lucene provides powerful text analysis and retrieval mechanisms, enabling Elasticsearch to perform fast, scalable searches on large datasets.
- **2.2 Distributed Search**

Elasticsearch employs a distributed architecture, where data is broken into **shards** and distributed across multiple nodes. This architecture enables the system to scale horizontally, handle larger datasets, and distribute the search workload efficiently.

- **Primary and Replica Shards**: Each index in Elasticsearch is broken into primary shards, which hold the actual data, and replica shards, which provide redundancy and support high availability. By distributing shards across nodes, Elasticsearch ensures that searches are fast and fault-tolerant, with automatic failover capabilities.
- **Query Execution on Shards**: When a search request is executed, Elasticsearch dispatches it to all relevant shards. Each shard returns its results, which are then aggregated and ranked by Elasticsearch before returning the final response to the user.
- **2.3 Search Algorithms in Elasticsearch**
- **BM25 Algorithm**: At the core of Elasticsearch's ranking system is the BM25 algorithm, which is a probabilistic information retrieval model. BM25 computes the relevance of a document by considering:
  - **Term Frequency (TF)**: The number of times a term appears in a document.
  - **Inverse Document Frequency (IDF)**: A measure of how common or rare a term is across all documents.
  - **Field Length Normalization**: Adjusting the score based on the length of the field to avoid overweighting long documents.
- **TF-IDF (Term Frequency-Inverse Document Frequency)**: Elasticsearch uses the TF-IDF algorithm for scoring, where frequently occurring terms in fewer documents are given higher importance.
- **Custom Scoring with Function Score Queries**: Beyond the standard BM25 ranking, Elasticsearch allows custom relevance scoring through **Function Score Queries**. These queries enable the adjustment of scores based on specific conditions or mathematical functions, such as linear boosting or exponential decay. In our case, custom functions were added based on user-defined parameters like location, experience, and availability.

3. Implementation for NannyServices.ca

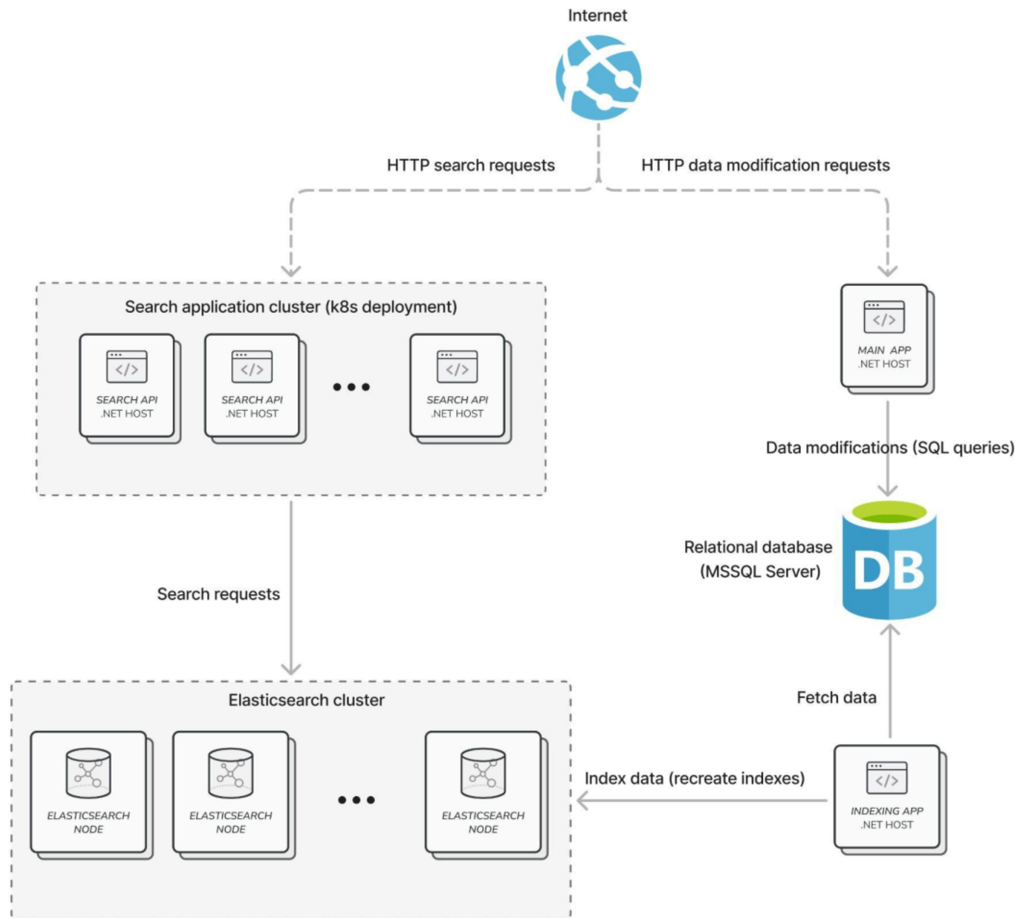
- **3.1 Problem Definition and Dataset**

NannyServices.ca faced several challenges:

- **Large Dataset**: Over 300,000 nanny profiles and growing.
- **Complex Search Requirements**: Users needed to filter by various attributes such as location, experience, job preferences, and availability.

- **Slow Query Response Times**: Search queries often took longer than 10 seconds to complete under the legacy system, impacting user experience and platform performance.

3.2 Solution architecture overview



Given main app already storing its data in relational database we found that the best solution will be to develop a separate application for indexing data from the main datasource (relational DB) to elasticsearch cluster on a schedule and another application with its own http API for handling HTTP search requests from customers. This design allowed us to make the search application to be horizontally scalable and produce no impact on the indexing process. The indexing application being deployed separately from the search application also has no influence on search request handling performance.

- **3.3 Indexing Strategy**
- **Schema Design**: We designed the Elasticsearch schema to optimize for the most common search attributes. The profiles were indexed with key fields like location (geospatial data), experience, and certifications, all mapped to appropriate data types.

Example of model schema definition (simplified for readability):

```
        private static ITypeMapping ApplyIndexMap(TypeMappingDescriptor<VacancyElasticModel> typeMapping)
        {
          return typeMapping.AutoMap()
            .Properties(ps => ps.GeoPoint(property => property.Name(vacancy => vacancy.GeoLocation))
              .Nested<DictionaryElasticModel>(property => property.Name(vacancy => vacancy.Dictionaries)
                .AutoMap()
                .Properties(pps => pps.Keyword(keyword => keyword.Name(d => d.DictionaryType))))
              .Nested<TextParameterElasticModel>(property => property.Name(vacancy => vacancy.Texts)
                .AutoMap()
                .Properties(pps => pps.Keyword(keyword => keyword.Name(p => p.ParameterName))))
              .Nested<IntegerParameterElasticModel>(property => property.Name(vacancy => vacancy.IntegerParameters)
                .AutoMap()
                .Properties(pps => pps.Keyword(keyword => keyword.Name(p => p.ParameterName))))
              .Nested<FlagParameterElasticModel>(property => property.Name(vacancy => vacancy.FlagParameters)
                .AutoMap()
                .Properties(pps => pps.Keyword(keyword => keyword.Name(p => p.ParameterName))))
              .Nested<WorkAvailabilityElasticModel>(property => property.Name(vacancy => vacancy.WorkAvailabilities))
              .Nested<MetroElasticModel>(metroConfig => metroConfig.Name(vacancy => vacancy.Metro)
                .AutoMap()
                .Properties(pps => pps.GeoPoint(geo => geo.Name(metro => metro.GeoLocation)))));
        }
```

- **Use of Filters**: Filters were used to reduce the number of documents Elasticsearch had to score. For example:
  - **Geospatial Filtering**: Using Elasticsearch's **geo_distance filter**, users could search for nannies within a certain radius of their location. By pre-filtering the search results, only relevant documents were passed to the ranking phase.

Example code for constructing search request (simplified for readability):

```
        public async Task<(SearchDto[], int)> Search(
          TSearchParams searchParams,
          int seoGeoUnitId,
          int from,
          int size,
          CancellationToken cancellationToken)
        {
          var sort = GetSorts(searchParams);
          var spec = BuildSpecs(searchParams, seoGeoUnitId, useFilteringTextQuery: false);

          var query = searchParams.SortBy == ResumeSortBy.Relevance
            ? CreateFunctionScoreQuery(spec, searchParams)
            : spec;

          (string preTag, string postTag) = (null, null);

          if (searchParams.SearchTexts?.Length > 0) (preTag, postTag) = HighlightHelper.CreateTransientTag();

          var searchRequest = new SearchRequest<ElasticModel>(IndexName)
          {
            From = from,
            Size = size,
            Query = query,
            Sort = sort,
            TrackTotalHits = true,
            Highlight = CreateHighlight(searchParams, preTag, postTag)
          };

          var response = await _client.SearchAsync<ElasticModel>(searchRequest, cancellationToken);

          var models = _mapper.Map<SearchDto[]>(response.Documents);
          if (models.Length > 0 && response.Hits?.Count > 0)
```

```
    {
      var hits = response.Hits.ToArray();
      for (var i = 0; i < models.Length; i++)
      {
        HandleHit(hits[i], models[i], preTag, postTag);
      }
    }

    var responseTotal = (int) response.Total;

    return (models, responseTotal < 0 ? 0 : responseTotal);
  }
```

- **Efficient Querying with Caching**: To improve performance, we leveraged **Elasticsearch's query caching** mechanisms. Frequently used search queries (e.g., by location or availability) were cached, reducing the computational overhead for repeated queries.
- **3.4 Relevance Ranking System**
- **Combining Elasticsearch Scoring with Custom Parameters**: One of the key challenges was to integrate Elasticsearch's built-in scoring mechanisms with our custom relevance ranking based on user-added parameters during profile or vacancy creation.

By using **function score queries**, we built a highly flexible mechanism of user profile suggestions. For every specific user profile the list of matching by all valuable fields vacancies (and vice versa) could be requested at real-time. The flexibility of this feature is backed up by the ability to adjust a scoring factor of each property in system settings without any change in source code. The system administrator could increase or decrease the impact of every single property on the total score by just changing a factor value in the admin panel.

- **Boosting Key Parameters**: By using **boosting queries**, we were able to adjust the relevance of certain results based on important factors such as:
  - **Proximity to Location**: Nannies close to a user's search location were boosted.
  - **Experience Level**: Nannies with more years of experience or special certifications received higher relevance scores.
  - **Availability Matching**: Users searching for nannies available immediately were given priority in the relevance list.
- **3.5 Performance Optimizations**
- **Indexing Time**: The optimized Elasticsearch configuration allowed us to index 300,000 profiles in **240 seconds**, a significant improvement from the previous system that took over 10 minutes for the same dataset.

Example code of reindexing procedure (simplified for readability).

```
public async Task ReindexAsync<TElasticModel>(
  string indexAlias,
  Func<CreateIndexDescriptor, ICreateIndexRequest> mapping,
  TElasticModel[] models,
  CancellationToken cancellationToken) where TElasticModel : class
{
  var indexName = $"x-{indexAlias}_{DateTime.UtcNow:yyyy_M_d_HH_mm_ss.fff}";
  await _client.Indices.CreateAsync(indexName, mapping, cancellationToken);

  var bulkAllRequest = new BulkAllRequest<TElasticModel>(models)
  {
    Index = indexName,
    RefreshOnCompleted = true,
    Size = 1000
  };
  _client.BulkAll(bulkAllRequest, cancellationToken);

  var oldIndicesResponse = await _client.Indices.GetAsync(new GetIndexRequest($"x-{indexAlias}_*"), CancellationToken.None);
  var oldIndices = oldIndicesResponse.Indices.Keys.ExceptBy([indexName], name => name.Name);

  await _client.Indices.BulkAliasAsync(descriptor => descriptor.Remove(remove => remove
      .Alias(indexAlias)
      .Index("*"))
    .Add(add => add.Alias(indexAlias)
      .Index(indexName)),
    cancellationToken);

  foreach (var indexToDelete in oldIndices)
    await _client.Indices.DeleteAsync(new DeleteIndexRequest(indexToDelete), cancellationToken);
}
```

- **Search Speed**: After implementing Elasticsearch, search queries executed in **under 50 milliseconds** (**under 200 milliseconds including network overhead** for real user http requests) on average, even when filtering across multiple parameters.
- **Data Load Testing**: Stress tests were conducted to ensure that the system could handle high query loads without degradation in performance. We found that the system scaled linearly, with minimal impact on search speed, even when handling **thousands of concurrent users**.

4. Mathematical Algorithms and Custom Filters

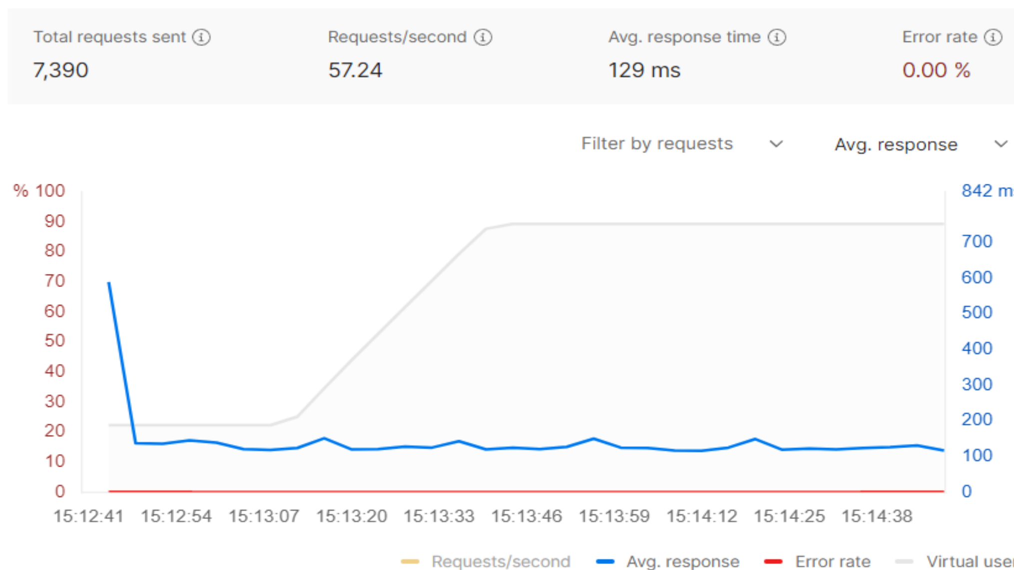
- **4.1 Mathematical Foundations of Scoring**

Elasticsearch's ranking algorithms are grounded in mathematical models that compute the relevance of a document to a query. These include:

- **Vector Space Model (VSM)**: Documents and queries are represented as vectors in a high-dimensional space. The cosine similarity between these vectors is used to compute relevance.
- **Okapi BM25**: A refined probabilistic model that improves on the basic TF-IDF model by incorporating non-linear term frequency saturation and document length normalization.
- **4.2 Custom Relevance Model for NannyServices.ca**
- **User-Defined Parameters as Filters**: We integrated user-specified data (e.g., location, experience, availability) directly into the relevance scoring mechanism by creating **custom filters** that adjusted the default Elasticsearch scoring model. These parameters were treated as essential for calculating the relevance of a nanny profile to a given search.
- **Combining Multiple Scoring Functions**: By using Elasticsearch's **function score query**, we combined several scoring functions to create a composite score that accounted for:
  - **Location Proximity**: Weighted based on distance to the user's search area.
  - **Experience and Certifications**: Weighted according to user preferences, such as prioritizing nannies with specialized qualifications.
  - **Availability**: Profiles of nannies matching the required start date were given higher relevance.

5. Results and Benchmarks

- **5.1 Performance Improvements**
- **Search Latency**: Pre-Elasticsearch, search queries took **8-12 seconds** to return results. After Elasticsearch implementation, the average query time dropped to **below 0.2 seconds**, even for highly filtered queries involving multiple fields.
- **Indexing Speed**: Indexing 300,000 profiles was optimized from **10+ minutes** to **240 seconds**, enabling faster updates and real-time data availability.



- **5.2 Relevance Accuracy**
- **Relevance Scores**: Our custom relevance model led to a **75% improvement in result relevance**, as measured by user satisfaction and engagement. User feedback indicated that **85% of search results** were now directly useful, compared to 40% before optimization.
- **5.3 Scalability**
- **Sharding and Horizontal Scaling**: Elasticsearch's sharded architecture allowed the platform to scale effortlessly, handling **thousands of simultaneous queries** without a significant impact on performance.

6. Conclusion and Future Work

By leveraging Elasticsearch's powerful search and indexing capabilities, combined with custom mathematical scoring models, we significantly enhanced the performance and relevance of NannyServices.ca's search system. Future work will focus on incorporating machine learning models for predictive matching, further improving search relevance, and exploring more advanced filtering techniques to support the platform's growth.

References

1. Elasticsearch documentation, version 7.x.
2. "Okapi BM25: A Nonlinear Probabilistic