Leveraging in-memory computation: Using Spark for textual queries

Presented by: Tejas Kannan
Date: 28/11/2018
Traditional Applications

Complex textual queries are generally expensive to run on traditional database platforms
Elasticsearch\textsuperscript{1} Background

Elasticsearch relies on inverted indexes to enhance search efficiency

1. winter is coming
2. yours is the fury
3. the choice is yours

<table>
<thead>
<tr>
<th>Term</th>
<th>Frequency</th>
<th>Documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>choice</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>coming</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fury</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>is</td>
<td>3</td>
<td>1,2,3</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>winter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>yours</td>
<td>2</td>
<td>2,3</td>
</tr>
</tbody>
</table>

\textsuperscript{1}Elasticsearch, https://www.elastic.co/
Example from: https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up
Elasticsearch requires...

...explicitly marking searchable fields at ingestion time

...dedicated index for each searchable field
Initial Elasticsearch Benchmark

Inverted indexes provide large performance improvements at the expense of additional storage

<table>
<thead>
<tr>
<th>Raw Data Size</th>
<th>Elasticsearch Size with Inverted Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.25 MB</td>
<td>56.73 MB</td>
</tr>
</tbody>
</table>

Notes on Experiment:
- 600,000 documents of actor/actress names from IMDb dataset\(^1\)
- Queries were 2 character strings based on common English names
- Error bars represent 25\(^{th}\)-50\(^{th}\)-75\(^{th}\) percentiles; data collected from 1000 trials

\(^1\)IMDb Dataset, https://www.imdb.com/interfaces/
Using Spark\textsuperscript{1} for Query Execution

Instead of requiring explicit indexes, we can try and use Spark as a computation engine for executing complex textual queries

Data Access Layer must use a persistent SparkContext to reduce job overhead


\textsuperscript{2}Redis, https://redis.io/
Why might using Spark + Redis be a good idea?

• FiloDB is an open-source database which uses Spark as a computation engine on top of Cassandra for real-time stream analysis\(^1\)

• Using Spark with Redis can provide over a 45x increase in performance over Spark + HDFS\(^2,3\)

• Spark as a computation engine provides flexibility of query execution

• By not requiring indexes for every searchable field, such a system can reduce the memory footprint

\(^{1}\)FiloDB, https://velvia.github.io/Introducing-FiloDB/
\(^{3}\)Redis Accelerates Spark by over 100 times, https://redislabs.com/press/redis-accelerates-spark-by-over-100-times/
Caching Intermediate Results

Using some additional memory, we can cache intermediate results to speed up future queries.

\[
\text{SELECT} \ * \ \text{FROM} \ \text{Hospitals} \ \text{WHERE} \ \text{name} \ \text{ends with} \ \text{“ity”}
\]
Caching Intermediate Results

Caching patterns can be chosen based on common phrases to maximize effectiveness with limited memory.

SELECT * FROM Hospitals WHERE name ends with “ity”

Redis Database \[\xrightarrow{T1: \text{filter}(h \Rightarrow h.name.endsWith(‘ty’))}\] RDD \[\xrightarrow{T2: \text{filter}(h \Rightarrow h.name.endsWith(‘ity’))}\] RDD

Cache of all documents with name ending with ‘ty’
Goals of this Project

• Create a Spark + Redis platform which can handle “prefix,” “suffix,” and “contains” queries

• Implement a caching feature using a configurable memory limit

• Benchmark the results against Elasticsearch to compare query latency and memory usage
Questions?
References

1. Elasticsearch, https://www.elastic.co/
2. Elasticsearch from the Bottom Up, https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up
4. IMDb Dataset, https://www.imdb.com/interfaces/