Distributed Indexing of Web Scale Datasets for the Cloud

Ioannis Konstantinou
Evangelos Angelou
Dimitrios Tsoumakos
Nectarios Koziris

Email:{ikons, eangelou, dtsouma, nkoziris}@cslab.ece.ntua.gr

Computing Systems Laboratory
School of Electrical and Computer Engineering
National Technical University of Athens
Greece
Problem

• Data explosion era: Increasing data volume (e-mail-web logs, historical data, click streams) pushes classic RDBMS to their limits
• Cheaper storage and bandwidth enables the growth of publicly available datasets.
  – Internet Archive’s WaybackMachine
  – Wikipedia
  – Amazon public datasets
• Centralized indices are slow to create and not scalable
Our contribution

• A Distributed processing framework to index, store and serve web-scale content under heavy request loads
• Simple indexing rules
• NoSQL and MapReduce combination:
  • MapReduce jobs process input to create index
  • Index and content is served through a NoSQL system
Goals 1/2

• Support of almost any type of datasets
  – Unstructured: plain text files
  – Semi-structured: XML, HTML
  – Fully structured: SQL Databases

• Near real-time query response times
  – Query execution times should be in the order of milliseconds
Goals 2/2

• Scalability (preferably elastic)
  – Storage space
  – Concurrent user requests

• Ease of use
  – Simple index rules
  – Meaningful searches
  – Find conferences whose title contains *cloud* and were held in *California*
• Raw dataset is uploaded to HDFS
• Dataset with index rules is fed to the Uploader, to create the Content table
• The Content table is fed to the Indexer that extracts the Index table
• The client API contacts the index table to perform searches, and the content table to serve objects
Index rules

Record boundaries

Attribute types

• Record boundaries split input into distinct entities used as processing units
• attribute types: record regions to index
  – A specific XML tag, HTML table, database column
• MapReduce class that crunches data input from HDFS to create the Content table
• Mappers emit a <MD5Hash, Hbase cell> key-value pair for each encountered record
Reducers lexicographically sort incoming key-values according to the MD5Hash.

Results are stored in HDFS in HFile format.

Hbase is informed about the new Content table.
Content table

<table>
<thead>
<tr>
<th>Row key: MD5Hash</th>
<th>Row value: record content</th>
</tr>
</thead>
<tbody>
<tr>
<td>2da0ae7cb598ac8e9455570a9c2f19fe</td>
<td><code>&lt;author&gt;&lt;name&gt;Ioannis&lt;/name&gt;&lt;surname&gt;Konstantinou&lt;/surname&gt;&lt;date&gt;20100424&lt;/date&gt;&lt;/author&gt;</code></td>
</tr>
<tr>
<td>223c14b2a8c7bbe24ba0d6854dd6f3cc</td>
<td><code>&lt;author&gt;&lt;name&gt;Evangelos&lt;/name&gt;&lt;surname&gt;Konstantinou&lt;/surname&gt;&lt;date&gt;20100426&lt;/date&gt;&lt;/author&gt;</code></td>
</tr>
</tbody>
</table>

• Acts as a record hashmap
• Each row contains a single record item
• Row key is the MD5Hash of the record content
• Allows fast random access reads during successful index searches
• Incremental content addition or removal
• Creates an inverted list of index terms and document locations (index table)
• MapReduce class that reads the Content table and creates the Index table
• Mappers read the content table and emit `<keyword,MD5Hash>` key-value pairs
• Reducers lexicographically sort incoming key-values according to the keyword value
• Reducers aggregate <keyword,MD5Hash> key-values to <keyword,list(MD5Hash)> key-values
• Results are stored in HDFS in HFile format
• Hbase is informed about the new Index table
Index table

<table>
<thead>
<tr>
<th>Row key: “term”_”attribute type”</th>
<th>Row value: list(MD5Hash)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20100424_date</td>
<td>2da0ae7cb598ac8e9455570a9c2f19fe</td>
</tr>
<tr>
<td>20100426_date</td>
<td>223c14b2a8c7bbe24ba0d6854dd6f3cc</td>
</tr>
<tr>
<td>Konstantinou_name</td>
<td>2da0ae7cb598ac8e9455570a9c2f19fe, 223c14b2a8c7bbe24ba0d6854dd6f3cc</td>
</tr>
</tbody>
</table>

- Row key: index term followed by the attribute type
- Row content: list of MD5Hashes of the records that contain this specific keyword in a specific attribute
Client API 1/3

- Built on top of HBase client API
- Basic search and get operations
- Google-style freetext queries over multiple indexed attributes
Client API 2/3

- Search for a keyword using Index table
  - Exact match (Hbase Get)
  - Prefix search (Hbase Scan).

- Retrieve object from Content table
  - Using a simple Hbase Get operation
• Supports AND-ing and OR-ing through client side processing
Experimental Setup

• 11 Worker Nodes, each with:
  – 8 cores @ 2GHz
  – 8GB Ram, disabled page swapping
  – 500GB storage

• A Total of 88 CPUs, 88GB of RAM and 5.55 TB of hard disk space

• Hadoop version 0.20.1

• Hbase version 0.20.2
  – Contributed 3 bug fixes for the 0.20.3 version
Hadoop-HBase Configuration

• Hadoop, HBase were given 1GB of RAM each
• Each Map/Reduce task was given 512MB of RAM
• Each worker node could concurrently spawn 6 Maps and 2 Reduces
• A total of 66 Maps and 22 Reduces
• Hadoop’s speculative execution was disabled
Datasets 1/2

• Downloaded from Wikipedia’s dump service and from Project Gutenberg’s custom dvd creation service

• **Structured:** 23 GB MySQL Database dump of the Latest English wikipedia
Datasets 2/2

• **Semi-structured** (XML-HTML)
  – 150 GB XML part of a 2.55TB of every English wiki page along with its revisions up to May 2008
  – 150 GB HTML from a static wikipedia version

• **Unstructured**: 20GB full dump of all languages of Gutenberg’s text document collection (46,000 files)
Real-life Query traffic

• Publicly available AOL dataset
  – 20M keywords
  – 650K users
  – 3 month period
• Clients created both point and prefix queries following a zipfian pattern
## Content table creation time

<table>
<thead>
<tr>
<th></th>
<th>XML</th>
<th>HTML</th>
<th>DB</th>
<th>TXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>size GB</td>
<td>time min</td>
<td>size GB</td>
<td>time min</td>
<td>size GB</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>44</td>
<td>10</td>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td>50</td>
<td>192</td>
<td>50</td>
<td>202</td>
<td>12</td>
</tr>
<tr>
<td>150</td>
<td>576</td>
<td>150</td>
<td>601</td>
<td>23</td>
</tr>
</tbody>
</table>

- Time increases linear with data size
- DB dataset is created faster than TXT
  - Less processing is required
Index Table size 1/2

- XML-HTML
  - Index growth gets smaller when dataset increases
  - For the same dataset size, XML index is larger than HTML index
• **TXT-DB**
  
  – TXT index is bigger for the same dataset size
  
  – Diversity of terms for the Gutenberg TXT dataset
Index Table creation time 1/2

- XML is more demanding compared to HTML
- A lot of HTML code gets stripped during indexing
Index Table creation time 2/2

- Time increases linear with data size
- DB dataset is created faster than TXT
  - Less processing is required
Indexing Scalability 23GB SQL

<table>
<thead>
<tr>
<th># of Nodes</th>
<th>time(min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

- The speed is proportional to the number of processing nodes
- Typical cloud application requirement
  - Extra nodes are acquired by a cloud vendor in an easy and inexpensive manner
Query Performance

• 3 types of queries
  – Point: keyword search in one attribute e.g. keyword_attribute
  – Any: keyword search in any attribute e.g. keyword_*
  – Range: Prefix query in any attribute e.g. keyword*

• Traffic was created concurrently by 14 machines

• Mean query response time for different
  – Load (queries/sec)
  – Dataset sizes
  – Number of indexed attribute types
Response time vs query load 1/2

- Range query loads above 140 queries/sec failed
- Response times for a load of 14 queries/sec
  - Point queries: 20ms
  - Any attribute queries: 150ms
  - Range queries: 27sec
• HBase caching result due to skewed workload
  – Up to 100 queries/sec there are enough client channels
  – Between 100 and 1000 queries/sec caching is significant
  – After 1000 queries/sec response time increases exponentially
Response time vs dataset size

- Average load of 14 queries/second
- Range queries are more expensive
- Response times for exact queries remain constant
Related Work 1/2

• MapReduce based data analysis frameworks
  – Yahoo’s PIG, Facebook’s Hive and HadoopDB
• Analytical jobs are described in a declarative scripting language
• They are translated in a chain of MapReduce steps and executed on Hadoop
• Query responses time in the order of minutes to hours
Related Work 2/2

• Distributed indexing frameworks based on Hadoop
  – Ivory distributes only the index creation but the created index is served through a simple DB
  – Hindex distributes Indices through HBase but the index creation is centralized

• In our system, both the index creation and serving is done in a distributed way
Questions