Large-Scale Search Engines and Language Technology

Aleksander Øhrn
alekso@microsoft.com

IFI, 2010.02.16
The Sweetspot

- Distributed Systems
- Information Retrieval
- Language Technology

[Logo: Secret Sauce Factory]
Web Search
alltheweb.com
1999-2003
Enterprise Search
Much more than intranets
Data Centers

alltheweb.com 2000
Data Centers
Microsoft 2010

http://www.youtube.com/watch?v=K3b5Ca6IzqE

http://www.youtube.com/watch?v=PPnoKb9fTkA
Search Platform Anatomy
The 50,000 Foot View

@

Crawler → Document Processing → Indexer

Data Mining

Index

Result Processing

Search

Query Processing

Front End
Scaling

- **Content Volume**
  - How many documents are there?
  - How large are the documents?

- **Content Complexity**
  - How many fields does each document have?
  - How complex are the field structures?

- **Query Traffic**
  - How many queries per second are there?
  - What is the latency per query?

- **Update Frequency**
  - How often does the content change?

- **Indexing Latency**
  - How quickly must new data become searchable?

- **Query Complexity**
  - How many query terms are there?
  - What is the type and structure of the query terms?
Scaling

Query Traffic

Scale through replicating the partitions

Content Volume

Scale through partitioning the data
Crawling The Web
Processing The Content

- HTML, PDF, Word, Excel, PowerPoint, XML, Zip, ...
- UTF-8, ISCI, KOI8-R, Shift-JIS, ISO-8859-1, ...
- English, Polish, Danish, Japanese, Norwegian, ...
- Title, headings, body, navigation, ads, footnotes, ...

- Format detection
- Encoding detection
- Language detection
- Parsing

- “30,000”, “L’Hôpital’s rule”, “台湾研究”, ...
- Øhrn, Ohrn, Oehrn, Òhrn, ... 
- Go, went, gone 
  Car, cars 
  Silly, sillier, silliest 
- “buljongterning”, “Rindfleischetiketterungsüberwachungsaufgabenübertragungsgesetz”, ...

- Tokenization
- Character normalization
- Lemmatization
- Decomounding

- Persons, companies, events, locations, dates, quotations, ...
- Who said what, who works where, what happened when, ...
- Positive or negative, liberal or conservative, ...
- Sports, Health, World, Politics, Entertainment, Spam, Offensive Content, ...

- Entity extraction
- Relationship extraction
- Sentiment analysis
- Classification
Creating The Index

<table>
<thead>
<tr>
<th>Word</th>
<th>Document</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>tea</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>teacart</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>teach</td>
<td>2</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>233</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>77</td>
</tr>
<tr>
<td>teacher</td>
<td>2</td>
<td>57</td>
</tr>
</tbody>
</table>
Deploying The Index
Processing The Query

- "LED TVs between $1000 and $2000"
- "I am looking for fish restaurants near Majorstua"
- "brintney speers pics"
- "23445 + 43213"
- "hphotos-snc3 fbcdn"
Searching The Content

Recall basic merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries

If the list lengths are $m$ and $n$, the merge takes $O(m+n)$ operations.

Can we do better?
Yes (if index isn’t changing too fast).

Augment postings with skip pointers
(at indexing time)

Why?
- To skip postings that will not figure in the search results.
- How?
- Where do we place skip pointers?


Assess relevancy as we go along
Searching The Content

Federation
Query processing
Result processing

Dispatching
Merging

Searching
Caption generation

“Divide and conquer”
Searching The Content

Tiering

• Organize the search nodes in a row into multiple tiers

• Top tier nodes may have fewer documents and run on better hardware

• Keep the good stuff in the top tiers

• Only fall through to the lower tiers if not enough good hits are not found in the top tiers

• Analyze query logs to decide which documents that belong in which tiers

“All search nodes are equal, but some are more equal than others”
“If the result set is too large, only consider the superior contexts”
Relevancy

Basic statistics
- Term frequency, inverse document frequency, completeness in superior contexts, proximity, ...

Document quality
- Page rank, link cardinality, item profit margin, popularity, ...

Match context
- Title, anchor texts, headings, body, ...

Crowdsourced annotations
- Anchor texts, click-through queries, tags, ...

Timeliness
- Freshness, date of publication, buzz factor, ...

Relevancy score

“Maximize the normalized discounted cumulative gain (NDCG)”
Processing The Results

- **Faceted browsing**
  - What are the distributions of data across the various document fields?
  - “Local” versus “global” meta data

- **Result arbitration**
  - Which results from which sources should be displayed in a federation setting?
  - How should the SERP layout be rendered?

- **Unsupervised clustering**
  - Can we automatically organize the results set by grouping similar items together?

- **Last-minute security trimming**
  - Does the user still have access to each result?
Data Mining

MapReduce: Simplified Data Processing on Large Clusters
Jeffrey Dean and Sanjay Ghemawat
Google, Inc.

Abstract
MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real-world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The runtime system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programs without any experience with parallel and distributed systems to easily utilize the resources of a large-distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward: however, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures are common to all programming models that compute with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a reduction to each logical “record” in our input data: use a set of intermediate-key/value pairs, or applying a reduce operator to all the key/value pairs in a given day, etc. Most such computations are conceptually straightforward: however, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures are common to all programming models that compute with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a reduction to each logical “record” in our input data: use a set of intermediate-key/value pairs, or applying a reduce operator to all the key/value pairs in a given day, etc. Most such computations are conceptually straightforward: however, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures are common to all programming models that compute with large amounts of complex code to deal with these issues.

In this paper, we present a new declarative and composable computing language, SCOPE (Scalable Computing On Parallel Execution). SCOPE is designed to support the type of massive data analytics that arise in many domains: scientific research, commerce, social networking, healthcare, and more. SCOPE is a high-level language for expressing parallel programs that run on large clusters. SCOPE programs are executed on a cluster of computing nodes, each of which is a commodity laptop or desktop computer. The nodes communicate with each other using a fast local network, and each node runs its own operating system.

SCOPE programs are expressed as a set of data transformations, or “transformations.” Each transformation is a small piece of code that performs a specific operation, such as reading data from a file, writing data to a file, or performing a mathematical calculation. Transformations are executed in parallel on different nodes in the cluster. The overall program is constructed by connecting these transformations together using a graph of data dependencies.

In this paper, we present a new declarative computing language, SCOPE (Scalable Computing On Parallel Execution). SCOPE is designed to support the type of massive data analytics that arise in many domains: scientific research, commerce, social networking, healthcare, and more. SCOPE is a high-level language for expressing parallel programs that run on large clusters. SCOPE programs are executed on a cluster of computing nodes, each of which is a commodity laptop or desktop computer. The nodes communicate with each other using a fast local network, and each node runs its own operating system.

SCOPE programs are expressed as a set of data transformations, or “transformations.” Each transformation is a small piece of code that performs a specific operation, such as reading data from a file, writing data to a file, or performing a mathematical calculation. Transformations are executed in parallel on different nodes in the cluster. The overall program is constructed by connecting these transformations together using a graph of data dependencies.

In this paper, we present a new declarative computing language, SCOPE (Scalable Computing On Parallel Execution). SCOPE is designed to support the type of massive data analytics that arise in many domains: scientific research, commerce, social networking, healthcare, and more. SCOPE is a high-level language for expressing parallel programs that run on large clusters. SCOPE programs are executed on a cluster of computing nodes, each of which is a commodity laptop or desktop computer. The nodes communicate with each other using a fast local network, and each node runs its own operating system.

SCOPE programs are expressed as a set of data transformations, or “transformations.” Each transformation is a small piece of code that performs a specific operation, such as reading data from a file, writing data to a file, or performing a mathematical calculation. Transformations are executed in parallel on different nodes in the cluster. The overall program is constructed by connecting these transformations together using a graph of data dependencies.
Applications
Spellchecking

http://www.google.com/jobs/britney.html
Spellchecking

1. Generate a set of candidates per query term using approximate matching techniques. Score each candidate according to, e.g., “distance” from the query term and usage frequency.

2. Find the best path in the lattice using the Viterbi algorithm. Use, e.g., candidate scores and bigram statistics to guide the search.
Entity Extraction

1. Logically annotate the text with zero or more computed layers of meta data. The original surface form of the text can be viewed as trivial meta data.

2. Apply a pattern matcher or grammar over selected layers. Use, e.g., handcrafted rules or machine-trained models. Extract the surface forms that correspond to the matching patterns.
Sentiment Analysis

1. To construct a sentiment vocabulary, start by defining a small seed set of known polar opposites.

2. Expand the vocabulary by, e.g., looking at the context around the seeds in a training corpus.

3. Use the expanded vocabulary to build a classifier. Apply special heuristics to take care of, e.g., negations and irony.
Contextual Search

“Sentences where someone says something positive about Adidas.”

“Paragraphs that discuss a company merger or acquisition.”

“Paragraphs that contain quotations by Alan Greenspan, where he mentions a monetary amount.”

“Sentences where the acronym MIT is defined.”

“Dates and locations related to D-Day.”

Persons that appear in documents that contain the word {soccer}

<table>
<thead>
<tr>
<th>person@base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack Nicklaus (~10.0%)</td>
</tr>
<tr>
<td>Fred Davis (~10.0%)</td>
</tr>
<tr>
<td>Billie Jean King (~8.0%)</td>
</tr>
<tr>
<td>Richard Nixon (~8.0%)</td>
</tr>
<tr>
<td>John Wayne (~7.0%)</td>
</tr>
<tr>
<td>Margaret Smith (~7.0%)</td>
</tr>
<tr>
<td>Joe Frazier (~7.0%)</td>
</tr>
<tr>
<td>Inna Rodina (~7.0%)</td>
</tr>
<tr>
<td>Nao Zedong (~6.0%)</td>
</tr>
<tr>
<td>Gordei Howe (~6.0%)</td>
</tr>
<tr>
<td>Richard M. Nixon (~5.0%)</td>
</tr>
</tbody>
</table>

Persons that appear in paragraphs that contain the word {soccer}

<table>
<thead>
<tr>
<th>person@base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diego Maradona (~4.0%)</td>
</tr>
<tr>
<td>David Beckham (~4.0%)</td>
</tr>
<tr>
<td>Alan Shearer (~3.0%)</td>
</tr>
<tr>
<td>Michelle Akers (~3.0%)</td>
</tr>
<tr>
<td>Mia Hamm (~3.0%)</td>
</tr>
<tr>
<td>Eric Wynalda (~3.0%)</td>
</tr>
<tr>
<td>Freddy Adu (~3.0%)</td>
</tr>
<tr>
<td>Michel Platini (~2.0%)</td>
</tr>
<tr>
<td>Stanley Matthews (~2.0%)</td>
</tr>
<tr>
<td>Oliver Neuville (~2.0%)</td>
</tr>
<tr>
<td>Bobby Moore (~2.0%)</td>
</tr>
</tbody>
</table>

Example from Wikipedia
Contextual Search

1. During content processing, identify structural and semantic regions of interest. Mark them up in context, possibly decorated with meta data.

2. Make all the marked-up data fully searchable in a way that preserves context and where retrieval can be constrained on both structure and content. Possibly translate natural language queries into suitable system queries.

3. Aggregate data over the matching fragments and enable faceted browsing on a contextual level.
Machine Translation
Query Completion
Caption Generation

- **Intra-document search**
  - Locate and rank relevant document fragments
  - But do it fast!

- **Perceived relevancy**
  - First impressions count
  - Can make or break a service

- **Trends towards richer captions**
  - Format-specific interactivity
  - Actionable elements
The Future