Distributed Indexing for Semantic Search

Peter Mika
Yahoo Research
Avinguda Diagonal 177
Barcelona, Spain
pmika@yahoo-inc.com

ABSTRACT

In this paper we describe the process of building indices for semantic search using MapReduce. We compare the two most straightforward representations of RDF data, the horizontal index structure using parallel indices and the vertical index structure using fields. We measure the cost of building indices and also compare retrieval performance on keyword queries and queries restricted to particular properties.

1. INTRODUCTION

In the last two years, the Semantic Web has been steadily growing in size, mostly as a result of the Linked Data effort and a renewed interest in annotating web pages using microformats and RDFa. Our own estimates put the size of the Semantic Web over 100 billion triples. The amount of data to be indexed by semantic search engines means that single machine solutions are not sufficient any more, and indexing has to be done in a distributed fashion in order to be efficient. In this short paper we look at the efficiency of building index structures using Hadoop, the open source implementation of MapReduce that is increasingly becoming the standard framework of choice in cloud computing. We implement the two most common ways of indexing RDF data, i.e. horizontal indexing using two parallel indices and vertical indexing using one field per property.[1] We compare these two methods by measuring the cost of indexing as well as their performance in retrieval, and also discuss their limitations in scalability.

2. RELATED WORK

Although most attention in the Semantic Web community has focused on building triple stores with expressive query languages and using database technology, there are a small number of Semantic Web search engines that implement large-scale search using inverted indices as studied in Information Retrieval. The best known example Sindice and its open source IR engine Siren [6], which builds on Lucene, the popular Java IR package. There are also a number of works on the core topic of ranking for semantic search, where the authors typically provide their own implementations of index structures to fit the query needs of a particular scenario, see e.g [3, 4, 7, 8]. In most cases, researchers build on Lucene and rely on single machine indexing for their experiments.

The general idea of using MapReduce for distributed index generation is described in modern textbooks such as [5]. In fact, indexing has been one of the examples given in the original description of Google's version of MapReduce by Dean and Ghemawat [2]. Hadoop is an open source implementation of MapReduce. It has been used in a relatively straightforward manner by the open-source Katta project[2] for generating text indices for Lucene.

Our focus is on large scale indexing to support the basic query functionality typically expected from Semantic Web search engines. To our knowledge, our work is the first detailed description of building index structures for RDF data using MapReduce and comparing possible alternatives with respect to the efficiency of both index construction and retrieval performance.

3. INDEXING RDF DATA

The index structures that need to be built for any particular search engine are largely determined by the required query functionality. In the case of semantic search engines, the basic functionality expected from (and provided by) search engines is the ability to retrieve resources based on words that must appear in the values of properties, possibly restricted by the name of the property where the words should appear.

Examples include looking for resources that contain the words *peter mika barcelona*, possibly made more precise by restricting occurrences to certain fields, e.g. *foaf:name = "peter mika" vcard:location = "barcelona"* in one possible query representation, where foaf:name stands for the name property from the FOAF ontology and vcard:location stands for the location property from the VCard-in-RDF ontology.

The two fundamental ways of achieving this functionality is by using either *parallel indices* or *fields*, functionalities commonly present in existing retrieval packages such as Solr or MG4J. The first option is illustrated in Table 1. For simplicity, we will call this a *horizontal index* on the basis that RDF resources are represented using only two fields, one field for the tokens and one for the properties. This is a parallel index in that there is a correspondence between the positions in the different fields, i.e. the value in the token field at a given position is (part of) the value for the property written to the same position in the property index. In order to perform the queries above, the query language needs to

[1] This is not to be confused with horizontal and vertical partitioning of index structures.

WWW2010, April 26-30, 2010, Raleigh, North Carolina.

[2] `http://katta.sourceforge.net/`

Field	p1	p2	p3	p4
token	peter	mika	32	barcelona
property	foaf:name	foaf:name	foaf:age	vcard:location

Table 1: Horizontal indexing of RDF data

Field	p1	p2	p3	p4
foaf:name	peter	mika		
foaf:age	32			
vcard:location	barcelona			

Table 2: Vertical indexing of RDF data

provide the *alignment operator*, which restricts matches in a certain field based on what appears at the same position in another field of the index.

The second option, which we will call a *vertical index* and show in Figure 2, is to create a field for each property occurring in the data. In this case performing our structured queries only requires the ability to restrict matches by field. Positions can be still useful, e.g. to make sure the first and last name are matched as consecutive words.

4. DISTRIBUTED INDEXING USING MAPREDUCE

The MapReduce paradigm is an ideal fit for the task of building basic inverted indices. The map phase can be used to load documents from the distributed file system and parse them into (key,value) pairs corresponding to terms and documents respectively. The first step in the reduce part of the framework performs precisely what is required to "invert", i.e. to collect all values belonging to the same key, corresponding to the documents that contain the same term. This is the input to the user-provided reduce function, which can write the index to the distributed file system, e.g. by using the indexing component of any IR package designed for single machine indexing. In our case, we use MG4J.The resulting (sub-)indices, where each index contains a subset of the dictionary, need to be merged separately, a functionality that MG4J provides.

On top of this basic scheme, we need two simple extensions to implement positions and fields. When positions are required, the value that is passed between mapper and reducer becomes a pair of document and position. For implementing fields, we need to capture the field identifier, and the key itself becomes a pair of term and field id. This is straightforward to do in Hadoop, where both keys and values can be arbitrary Java objects as long as they can be serialized.

Though all of the above is easy to implement, there are two more practical requirements that are somewhat more difficult to address. The first is that the values need to be sorted, because indexing requires to store occurrences in increasing order of document identifiers and positions. Although it is possible to sort values within the reduce function itself, the size of the memory would limit the maximum number of occurrences per term. (In particular, memory would run out for the most popular terms.) Fortunately, there is a way to implement a secondary sort on values by rewriting part of the sort operation that precedes the reduce, which is performed on disk. The key idea is to make the value a part of the key, and then rewrite the sort function to take the values into account. The partitioning of data among reducers and the grouping of data on reducer nodes need to be rewritten as well, because these operations are now done on only a part of the key. The price to pay is an increase in the amount of data that needs to be passed between mappers and reducers.

The second requirement for indexing (at least in the case of MG4J) is that the number of documents needs to be known at the beginning of writing out occurrences for a given term. This is a problem, because Hadoop only provides an iterator on the values, and the developers has no way of knowing how many values are there to read. This problem is non-trivial, and even more complex when considering positions, since we need to know the number of documents, not the number of occurrences (document, position pairs) that will be read. Our solution is again to trick the framework, in this case by introducing a dummy occurrence for every term and document pair that is being indexed. These occurrences have a document id that is set to -1, which is less than the document id of the first real document, and therefore they are sorted to appear as the first in the list of values. Counting these dummy occurrences tells us how many documents will be read. We again pay the price of increased load, although there will be only one dummy occurrence per document and term combination, which is much less than the total number of (key, value) pairs when considering positions.

5. EXPERIMENTS

We have implemented distributed indexing using Hadoop as described above, and using MG4J as the underlying search engine.

For our experiments, we have used the "urified" version of the Billion Triples Challenge data set from 2009, which is available for download in NTuples format.[3] For indexing, we have pre-grouped the data by subject, i.e. collecting all the triples with the same subject. The uncompressed size of this input data is 247GB. This includes all triples, although only the triples literal valued objects were indexed. To further reduce the size of the data to be indexed, we have ignored subjects with more than 10KB data. We have also ignored words that appeared on our blacklist of 389 stopwords. Lastly, we ignored words consisting entirely of numbers. While this clearly should not be done in such an indiscriminate manner, it helped to reduce the dictionary size, which is otherwise bloated by numbers.

To further increase efficiency, we have encoded URIs using MG4J's `LcpMonotoneMinimalPerfectHashFunction`, which provides a hash function that is minimal, perfect and monotone, and given sufficient memory can be efficiently built even on a single machine [1]. URI encoding is critical in reducing the amount of data that needs to be processed at both indexing and retrieval time. In our case, we had to encode 127 million URIs (of 9.7GB total size in plain text). The resulting hash function occupies only 307MB on disk, i.e. uses only 2.4 bytes per URI. We distribute these files to mappers using Hadoop's DistributedCache mechanism (reducers do not need access to the hash function) and due to its small size we are able to keep it in memory.

Indexing our data taught us a number of valuable lessons.

[3] `http://km.aifb.uni-karlsruhe.de/ws/dataset_semsearch2010/000-CONTENTS`

Scheme	Indexed URIs	Indexed triples	Occurrences	Size before merging	Merged index size
Horizontal	114,530,196	273,922,563	2,931,625,024	4.554 MB	8,948 MB
Vertical	114,530,196	262,564,786	1,438,318,071	5,195 MB	10,779 MB

Figure 1: Index statistics

First, despite the relative maturity and user friendliness of Hadoop, fine tuning the process of indexing was non-trivial especially when configuring memory usage. Hadoop itself has a large number of parameters that determine how much memory is used for various parts of the framework, and these critically affect how much memory is left for the user. In most cases, there is a real trade-off between system performance and free memory for processing. The second set of experiences relate to the scalability of indexing. One clear bottleneck arises when the data is overly skewed in the distribution of properties, i.e. when certain properties are much more commonly used in the data than others. This causes an unbalanced distribution of load among reducers, with the effect that some reducers will end up with too much data to shuffle, sort and reduce. Another limitation affects the vertical scheme: the number of fields that we could index was limited by the size of the available memory on the compute nodes. The reason is that for efficient disk usage MG4J itself needs to cache data in memory separately for each field. This is a hard constraint. The solution we took was to limit the indexing to the top 300 datatype properties for the vertical scheme for a case where reducers had roughly 2GB of available memory. Although we have not done so, we could have built indices for additional fields by taking multiple passes at the data.

5.1 Indexing performance

Table 1 shows the overall statistics of the indexed data and the resulting indices. Per above, we index less triples in the vertical scheme because only the most frequent properties are indexed. Still, due to the dominance of these top properties, we loose less than five percent of the triples, and presumably triples with predicates that are less likely to be searched for (since they are less often provided).

We report the execution times and other key performance metrics in Table 2. We note that reporting accurate runtimes using Hadoop is problematic. The runtime reported by the system is a measure of the execution time from start to finish, which is influenced by a number of factors, in particular cluster load. Depending on the availability of a cluster the individual map and reduce tasks may spend different periods of time in a pending state, i.e. waiting to be executed. Therefore we report the runtime as an indication only. Using the latest version of Hadoop, we can obtain the time that mappers and reducers spent in execution in aggregate (`SLOTS_MILLIS_MAPS` and `SLOTS_MILLIS_REDUCES`). However, this time will still include the time spent on failed tasks. (The larger the cluster, the more likely that individual jobs will fail, requiring a restart of that part of the processing.) Speculative execution, which is the ability to preemptively launch duplicates of slow or failing tasks, leads to further double counting. Lastly, it's problematic to compute the true runtime from these numbers because the map and reduce phases partly overlap.

Despite these caveats, it's clear that most of the time in execution is spent in the reduce phase. One minor issue with the map phase is that given a small cluster capacity,

Metric	Horizontal	Vertical
Real time	3h 18m	4h 51m
Maps	2,000	2,000
Time per map	166s	581s
Map output records	$4.018 * 10^9$	$2.627 * 10^9$
Map output size	144 GB	68 GB
Reduces	20	20
Time per reduce	8371s	14987s
Reduce shuffle bytes	42 GB	28 GB

Figure 2: Comparison of indexing efficiency for horizontal and vertical indexing

the high number of mappers —which is determined by the input size— can become a problem. For the current job size of 2000 mappers and considering two maps per node, for the ideal case we would need a cluster of 1000 machines. If that is not available, there will be less than ideal parallelism although most of the time will be still spent in the reduce phase. The number of mappers can be lowered by compressing the data, with some cost of uncompressing at runtime.

The time spent by reducers can be decreased by simply increasing the number of reducers, which is a user configurable parameter. The number of reducers also determines the number of subindices that will be generated. It is clear that with more and smaller subindices merging will get less efficient, although we have not yet explored the trade-off between indexing time and merging time.

5.2 Retrieval performance

We measure the performance of our indices by executing the same query set that is used for the Entity Search Track at the Semantic Search Workshop 2010. This query set contains 4497 queries sampled randomly from the queries that have been submitted at least three times to Yahoo's US search engine in January, 2009.[4]. We measure the time of the actual query execution, i.e. the duration of the `process()` call of the `QueryEngine` instance in MG4J. We focus on the relative performance of the various schemes; the absolute performance of the search engine may depend on many factors, in particular the hardware environment.

We assessed performance for two kinds of queries, keyword queries and unigram queries with a field restriction. For the latter, we have generated potential candidate queries from our data, by looking for matches of the query words within particular properties. It is future work to experiment with other query types or mixed loads generated to simulate some particular application scenario.

Figure 3 shows the convergence of query execution times for keyword queries. The average query execution time in the horizontal scheme was around 80 ms in comparison to around 1s for the vertical scheme, i.e. the horizontal scheme

[4]This dataset is available as part of the Yahoo! Webscope program, see `http://webscope.sandbox.yahoo.com/`

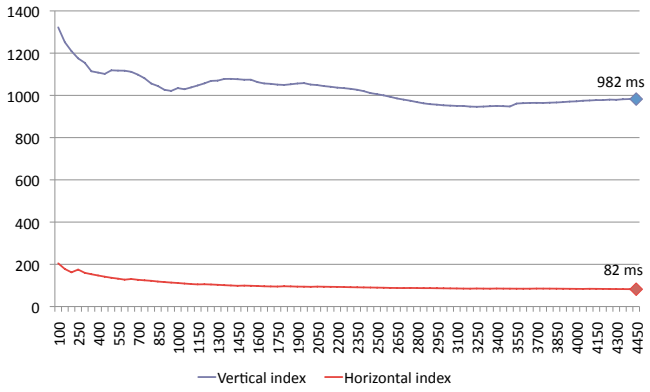

Figure 3: Retrieval performance on keyword queries

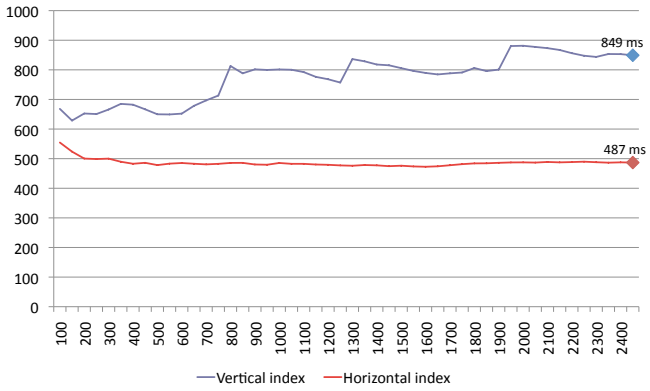

Figure 4: Retrieval performance on unigram queries with property restriction

performed 10-12 times better. This can be explained by the physical storage of fields: for keyword queries without field restrictions, a separate disk-seek is required to inspect the contents of the each field.

Similarly, Figure 4 shows the convergence of query execution times for unigram queries with property restriction, i.e. queries of the form *foaf:name = mika*. Somewhat surprisingly, the horizontal scheme seems to outperform the vertical scheme, which we can not fully explain at this point. The convergence for the vertical scheme is slow: field restrictions produce very uneven execution times. Execution times vary from a few milliseconds to several seconds. The explanation is that queries on less frequently used fields or fields with smaller values are more efficient to evaluate. We plan to repeat these experiments with larger numbers of queries to achieve more solid convergence.

6. DISCUSSION

In this paper we have described in some detail the implementation of building index structures for RDF data using Hadoop's MapReduce. The presented work provides a preliminary analysis of the efficiency of distributed indexing using MapReduce and retrieval performance using various index structures. In future work, we may look at

- Additional query types such as joins, and hybrid queries, i.e. queries containing keywords with and without property restriction
- Indexing object properties in addition to datatype properties
- Different collections, in particular Linked Data versus embedded metadata (RDFa, microformats)
- CPU usage for different query loads and index structures

Last, but not least we are looking for additional ways of improving the performance of both index generation and retrieval, for example by exploiting known properties of the query stream to be evaluated.

Acknowledgement

The author would like to thank Roi Blanco for his advice on this paper.

7. REFERENCES

[1] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with $O(1)$ accesses. In C. Mathieu, editor, *SODA*, pages 785–794. SIAM, 2009.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[3] A. Duke, T. Glover, and J. Davies. Squirrel: An advanced semantic search and browse facility. In *ESWC*, pages 341–355, 2007.

[4] M. Fernandez, V. Lopez, M. Sabou, V. Uren, D. Vallet, E. Motta, and P. Castells. Semantic Search Meets the Web. In *IEEE Semantic Computing*, pages 253–260, 2008.

[5] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[6] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tummarello. Sindice.com: a document-oriented lookup index for open linked data. *IJMSO*, 3(1):37–52, 2008.

[7] C. Rocha, D. Schwabe, and M. P. Aragao. A hybrid approach for searching in the semantic web. In *Proceedings of the 13th international conference on World Wide Web*, pages 374–383. ACM Press, 2004.

[8] H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan. Semplore: A scalable ir approach to search the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):177 – 188, 2009. The Web of Data.