Web-scale Entity Annotation Using MapReduce

Shashank Gupta
IIT Bombay
shashank91.bits@gmail.com

Varun Chandramouli
NetApp
varun.c37@gmail.com

Soumen Chakrabarti
IIT Bombay
soumen@cse.iitb.ac.in

***Abstract*—Cloud computing frameworks such as map-reduce (MR) are widely used in the context of log mining, inverted indexing, and scientific data analysis. Here we address the new and important task of annotating token spans in billions of Web pages that mention named entities from a large entity catalog such as Wikipedia or Freebase. The key step in annotation is disambiguation: given the token *Albert*, use its mention context to determine which Albert is being mentioned. Disambiguation requires holding in RAM a machine-learnt statistical model for each mention phrase. In earlier work with only two million entities, we could fit all models in RAM, and stream rapidly through the corpus from disk. However, as the catalog grows to hundreds of millions of entities, this simple solution is no longer feasible. Simple adaptations like caching and evicting models online, or making multiple passes over the corpus while holding a fraction of models in RAM, showed unacceptable performance. Then we attempted to write a standard Hadoop MR application, but this hit a serious load skew problem (82.12% idle CPU). Skew in MR application seems widespread. Many skew mitigation approaches have been proposed recently. We tried SkewTune, which showed only modest improvement. We realized that reduce key splitting was essential, and designed simple but effective application-specific load estimation and key-splitting methods. A precise performance model was first created, which led to an objective function that we optimized heuristically. The resulting schedule was executed on Hadoop MR. This approach led to large benefits: our final annotator was 5.4× faster than standard Hadoop MR, and 5.2× faster than even SkewTune. Idle time was reduced to 3%. Although fine-tuned to our application, our technique may be of independent interest.**

Keywords: MapReduce; Hadoop; Data Skew; Partitioning; Web entity annotation.

1. Introduction

Thanks to automatic information extraction and semantic Web efforts, Web search is rapidly evolving [1] from plain keyword search over unstructured text to entity- and type-oriented queries [11], [23], [26] over semi-structured databases curated from Web text and online knowledge bases such as Wikipedia and Freebase.

As just one example of the vast potential of semantic search, if occurrences of the physicist Einstein are tagged in the Web corpus, and a knowledge base establishes a connection from the type scientist to Einstein, then we can directly return the *entity* Einstein, rather than "ten blue links" to pages, in response to the query `scientist played violin`. One can easily envisage the power of further combining such subqueries into complex queries, assembling tabular results, and computing aggregate statistics.

The key challenge on the way to semantic search is scalable and accurate annotation of entity mentions in Web text. There are many Einsteins and even more John Smiths. Considering a Web page as a sequence of tokens, given the tokens "Albert" or "Agent Smith" on a Web page, we need to use contextual information to judge which entity in our knowledge base, if any, is mentioned at those spans. This is usually done by sophisticated machine learning techniques [21], [6], [15], [14].

We are building CSAW [3], a Web-scale semantic annotation and search system. While there is no dearth of knowledge and public-domain code for harnessing commodity cluster computing toward indexing and querying text corpora, very little is known about the optimal use of, and benefit from, popular cluster-computing paradigms in Web-scale entity annotation. That is the focus of this paper.

1.1. Entity annotation background

CSAW uses a type and entity knowledge base, hereafter called the *catalog*. The type catalog is a directed acyclic graph of type nodes. Edges represent the transitive and acyclic *subTypeOf* relationship, e.g., Physicist *subTypeOf* Scientist. There are also entity nodes, e.g., there is a node for the Physicist Albert Einstein. Entity nodes are connected to type nodes by *instanceOf* edges. Each entity is mentioned in ordinary text using one of more *lemmas*. E.g., the city of New York may be called "New York", "New York City", or "Big Apple", and Albert Einstein may be called "Einstein", "Professor Einstein", or just "Albert". The relation between entities and lemma phrases is many-to-many. We will designate each lemma phrase in our catalog with an ID ℓ, and each entity with an ID e.

As we scan a document, we will encounter occurrences of lemmas in our catalog. Each such occurrence will be resolved to be a reference to some entity in our catalog, or rejected as not being in our catalog. E.g., there are many people named Einstein that are not in Wikipedia. This is basically a classification problem (where one class label is "reject"). To solve the classification problem for one instance of a lemma ℓ appearing in a certain position p of a document d, a classifier *model* M_ℓ needs to be loaded into RAM. The surrounding context of the occurrence of ℓ is abstracted and distilled into a context feature vector or CFV. Finally, the classifier reads the CFV and outputs zero or more likely entities, possibly with corresponding confidence scores. One such annotation record looks like (d, p, e) (ignoring the confidence score). These are later indexed into the form $e \rightarrow \{(d, p)\}$.

978-1-4799-0730-4/13/$31.00 ©2013 IEEE

1.2. The catalog scaling challenge

In the first edition of CSAW, our catalog contained about 2M entities from YAGO [27], a curated union of Wikipedia and WordNet. Even with this small number of entities, fitting M_ℓ for all ℓ simultaneously into the RAM of one host was a challenge [2]. But doing so helped us annotate the corpus at the greatest possible speed. Each host would load up all lemma models, and then stream through its share (partition) of the corpus. Any lemma encountered can be disambiguated immediately.

Query coverage of CSAW depends critically on annotating as many entities as possible, but YAGO registers only well-known entities. We are in the process of evolving our catalog from YAGO to include large parts of Freebase [8]. At the end of this process, we estimate we will have about 40M entities. At that point, one host's RAM will no longer be able to hold all lemma models. Even if we can somehow compress [2] the models to barely fit into RAM, this is a suboptimal use of RAM; we should rather reserve most of the RAM for index runs, so that the runs are larger and fewer run merges are needed in the end.

1.3. Our contributions

We begin, in Section 3, with the two simplest approaches to extend our system:

- pack subsets of models into RAM for successive passes over the corpus, and
- cache models and evict them if RAM is inadequate, while making a single pass over the corpus.

We present measured or reliably estimated running times and idle times with these approaches. Note that in both approaches, all, or a major portion of mentions in a document are disambiguated together in a batch.

Then, in Section 4, we propose a different paradigm that involves preparing, for each mention encountered, a self-contained record that includes all necessary context information to disambiguate the mention. This record is called a context feature vector or CFV. Before proceeding further, we make preliminary measurements on the basic complexity of this approach and show that it is competitive for our application.

Once the practicality of scattering CFVs is established, it is natural to explore a map-reduce framework [7]. This is done in Section 5, a central focus of this paper. We show that standard map-reduce implementations like Hadoop, and also recent skew mitigation techniques, will not match up to the workload offered by our application. Instead, we take control of policies for model replication and the assignment of models to processors. The result is a fully Hadoop-supported [12] Web-scale annotator that needs negligible amounts of RAM per cluster host and is 5.4× faster than standard Hadoop MR, and 5.2× faster than even SkewTune.

1.4. CSAW system and testbed

The CSAW [5] production system runs on 40 DL160G5 data and compute hosts, and two DL180G5 name nodes and NFS servers. Each host has 16 GB DDR2 RAM, eight 2.2GHz Xeon CPU cores, and two 1 TB 7200 rpm SATA hard drives allocated to Hadoop. Our production Web corpus is comparable to the ClueWeb12 [4] corpus, with 561,287,726 mostly-English pages. Each document, stripped of HTML tags, is stored compressed, with a size of about 3KB, for a total of about 1.6 TB.

When our annotator [2] is run on the corpus, on an average, 61 token spans per document are identified as potential references to one or more entities among about two million entities in YAGO [27]. Of these, on an average, 21 spans are actually associated with entities after disambiguation. Native CSAW (pre-Hadoop) takes about 0.6 milliseconds to disambiguate a spot, and such speed is essential to process billions of pages within practical time horizons.

Our final annotator described here can process the above corpus with high efficiency. However, some of the competing systems we have studied have much lower efficiency. To study many performance parameters within reasonable real time, we used a representative sample of the above corpus with 18 million documents, of total compressed size 54 GB. In a similar spirit, we held the entity catalog fixed as YAGO and simulated lemma model set scale-up (due to migration to Freebase) by shrinking RAM by the same factor, to stress-test all systems. For similar reasons, we ran most experiments with 20 of the 40 hosts, each using 4 cores.

2. Related work

Our approaches are inspired and guided by much recent work on skew mitigation in MR applications. We will discuss the most closely related work in detail in later sections. Here we provide a broad overview. The problem of data skew in MR is quite prevalent [25] and has been of significant interest for many researchers. Lin [19] highlights the problem of stragglers in MR due to Zipfian distribution of task times and how that places inherent limits on parallelism, when constrained by the requirement that all instances of the key must be processed together. He suggests that application-specific knowledge be used to overcome such efficiency bottlenecks. The central idea in this literature is to devise techniques to estimate costs of the application and modify the system to mitigate skew effects, both statically and dynamically [18].

Kwon et al. [16] have proposed SkewReduce, where the running time of different partitions depends on the input size as well as the data values. It uses an optimizer which is parametrized by user-defined cost function to determine how best to partition the input data to minimize computational skew.

SkewTune [17] is another approach which mitigates skew in MR applications dynamically by re-partitioning and re-assigning the unprocessed data allocated to a straggler to one of the ideal nodes in the cluster.

Gufler *et al.* [9], [10] suggest cost models for reducers as functions of the number of bytes and the number of records a reducer needs to process. Their algorithm splits reduce input data into smaller partitions, estimates their cost, and distributes

smaller partitions using two load balancing approaches-fine partitioning and dynamic fragmentation.

Whilst more general, these approaches do not mitigate skew to the best possible extent in cases where the constraint for a single reducer to process all the values corresponding to a key is relaxed.

In more recent work [24], the keys are divided among different partitions in order to obtain better load-balancing. Their cost model assumes uniform per-record processing time for different key-groups which turns out to be a very restrictive assumption for our application, as we shall see in the upcoming sections.

3. Simple adaptations

3.1. Bin packing

The simplest adaptation that minimizes code changes is to load a suitable subset of lemma models into RAM at a time, and make multiple passes through the corpus. We call this the *bin-packing* solution. Given each lemma model has size s_ℓ and available RAM of a host is of size S, we are looking to pack items $\{s_\ell\}$ into as few bins of size S as we can, to minimize the number of passes through the corpus.

A standard and effective heuristic for (the NP-hard problem of) bin packing is to sort models in decreasing size, pack them in a bin until full, then allocate the next bin.

The obvious disadvantage of this approach is that the CPU work needed to decompress and tokenize the corpus, and turn lemma occurrences into CFVs, is not nearly negligible, compared to the disambiguation work itself, and the former will be repeated, perhaps needlessly, several times. However, the simplicity of this scheme still makes it appealing to evaluate.

Total number of lemma models	1,685,856
Total size of lemma models	2.17 GB
Size of largest model	1.37 MB
Size of smallest model	13 B
Average size of model	1.35 KB

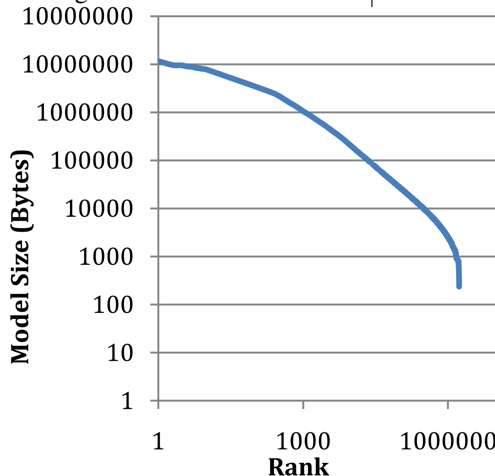

Figure 1. Distribution of lemma model sizes.

3.2. Bin packing performance

Figure 1 shows the distribution of lemma model sizes, along with some summary numbers. The distribution is quite skewed, the largest model being a thousand times larger than the average.

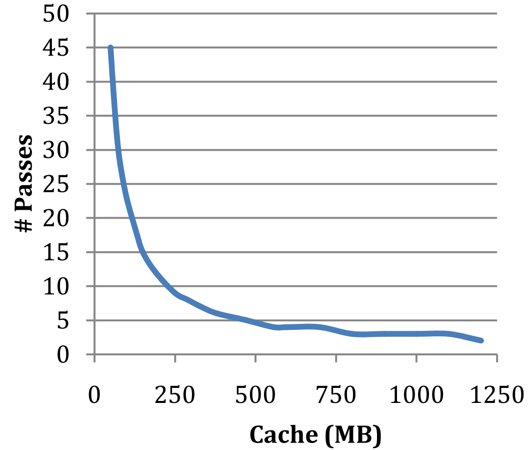

Figure 2. Number of corpus passes vs. cache size, using bin packing.

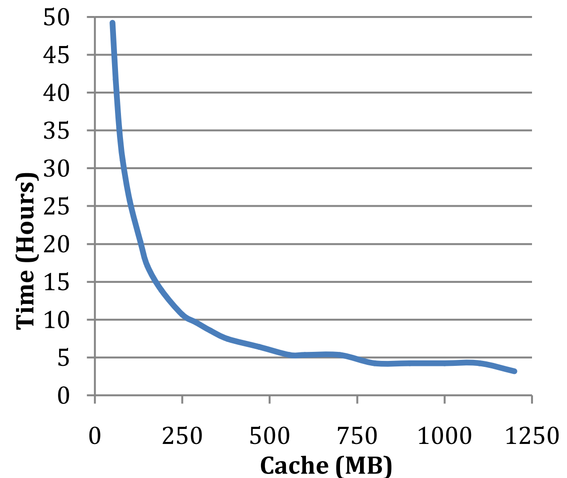

Figure 3. Total time vs. cache size, using bin packing.

Figures 2 and 3 show the effect of increasing cache sizes on the number of passes (bins) and the total time over all passes. To interpret these numbers in context, we estimate that incorporating Freebase will expand our total lemma model size from 2.17GB to at least 30GB, about a 15× growth. Therefore, as a reverse experiment, we can focus on the 100–200 MB range on the x-axis. I.e., we can calibrate our performance when hosts have 1/10th to 1/20th of the RAM needed for the current model size of 2.17GB. The number of passes in this range is 10–20, and this is directly reflected in the total running time. Given one pass over 500 million documents takes about a day (on 20 hosts, not one), this is completely unacceptable, especially so because much CPU work in decompressing, tokenizing, and searching for spots is needlessly repeated across the passes.

3.3. Caching disambiguation models

Not only do lemma models have diverse and highly skewed sizes, but the rates at which lemmas are encountered while scanning the corpus are also highly skewed [2]. This raises the hope that good hit rates and fast annotation may be achieved by maintaining a cache of lemma models within limited RAM, with a suitable model eviction policy such as least recently used (LRU) and least frequently used (LFU). However, one potential problem with caching is RAM fragmentation. In earlier work [2], models for all lemmas were carefully packed and compressed into one RAM buffer. Shattering that buffer into individual lemma models of diverse sizes, and repeatedly loading and unloading them from RAM, may cause intolerable stress on the memory manager and garbage collector of the JVM.

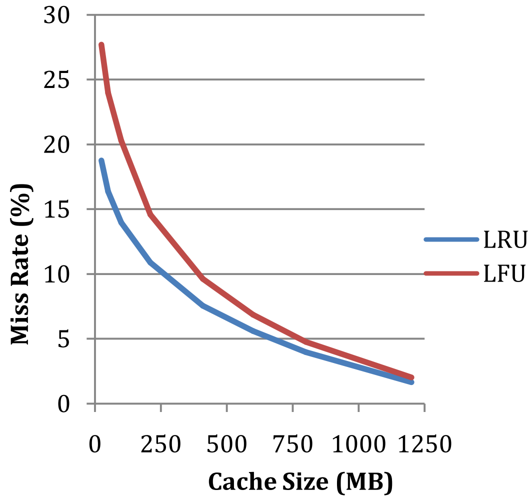

Figure 4. Cache miss rate vs. cache size.

3.4. Caching performance

Figure 4 shows miss rates for LFU and LRU as lemma model cache size is increased. LRU is superior. The absolute miss rates may look reasonably low (few percent). But this has to be reinterpreted in the light of the new application. Globally, about 284,000 CFVs are generated per second, about 14,200 CFVs at each of 20 hosts, as the corpus is scanned, tokenized, and lemmas are matched. Even a 10% miss rate means 1,421 misses per second per host. Even leaving aside for a moment the issue of cache memory management in the JVM and attendant garbage collection (GC), a miss almost certainly results in a disk seek (because the OS cache is overwhelmed by corpus scan and later, index run write-outs), which, on commodity 7,200 rpm disks, can easily cost 10 milliseconds. This makes miss servicing impossible even at 10% miss rates.

Figure 5 explores the sensitivity of the above findings to growth in the size of the model set. At a cache size of 200–400 MB, misses per second per host can double if the number of models is quadrupled. Therefore, extending from Wikipedia to Freebase relying on a caching approach is out of

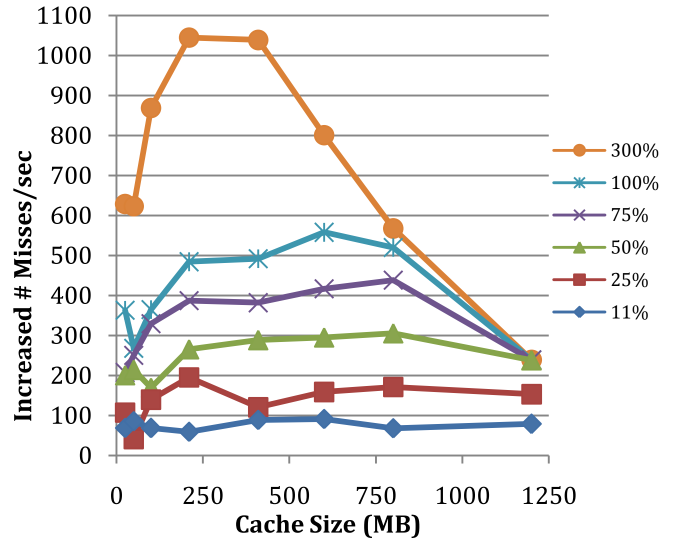

Figure 5. LRU miss rate change vs. percent increase in lemma model set.

the question — a larger catalog and richer features will only make matters worse.

Given the diverse sizes of models (Figure 1) loaded, evicted and reloaded, memory fragmentation and GC also presented insurmountable difficulties and led to impractical running times. Therefore we present just one data point: with 540 MB cache, the 54 GB corpus took 7.6 hours, compared to about 6 hours with bin packing given the same RAM.

3.5. Distributed in-memory model storage

At this point, our predictable reaction was to investigate the use of a distributed in-memory key-value store such as Memcached [20] or HBase [13] by storing lemma models into them (keyed by their ID), to see if we can avoid disk access on cache miss by converting it to an access over the network. Unless substantial tweaks (replication by hand, random key padding) are undertaken, only one host will handle request for a lemma key. Just to support the disambiguation of the most frequent lemma, the key-value store should be able to serve the corresponding model at the rate of 6.65 GB/s. Overall, to keep up with document scanning, tokenization, and detection of lemma matches, the key-value store should be capable of serving about 284,000 requests per second, involving about 69 GB of network transfer per second. (See Figure 6 for details.) These are all quite impractical on current commodity networks. Moreover, preliminary extrapolation suggests that quadrupling the number of lemma models will almost double the query bandwidth demanded from the key-value store. Therefore, matters will get much worse as we begin to recognize new lemmas from Freebase not currently in our catalog.

4. Scattering context feature vectors

Section 3 has made clear that retaining our earlier document-streaming form of annotation is not feasible. The

Lemma rank	Bandwidth	Queries/sec
1	6.65 GB/s	6,073
2	2.13 GB/s	2,641
3	2.11 GB/s	2,201
4	2.03 GB/s	2,107
5	2.01 GB/s	2,088
6	1.71 GB/s	2,036
7	1.37 GB/s	1,817
8	1.13 GB/s	1,509
9	1.12 GB/s	1,495
10	1.03 GB/s	1,471

Figure 6. Query and bandwidth demands by top lemmas on distributed key-value store.

other option is to perform the decompression, scanning, tokenization, detection of lemma occurrences, and conversion to CFVs exactly once, and thereafter work with CFVs alone, distributed suitably across the cluster. (Whether CFVs are instantiated to disk or not is a finer detail, depending on how disambiguation tasks are scheduled.) In this section, preparatory to applying MR (Section 5), we will evaluate and establish that the CFV scattering protocol is practical.

Each CFV initially has a key ℓ, its lemma ID. In the most general setting, the system installs a disambiguator for each lemma ℓ at one or more hosts, and CFVs keyed on ℓ are communicated over the network to one of these hosts, to get disambiguated. Different lemmas are encountered at diverse rates in the corpus. E.g., "John Smith" is far more frequent than "Apostoulos Gerasoulis". To address this skew, we may choose to more aggressively replicate M_ℓ for frequent ℓ to more hosts than rarer lemmas.

4.1. The global CFV shuffle

Consider all CFVs destined to one host. One option is to process them in the arbitrary order in which they are received, avoiding a per-host sort. In this case, as we disambiguate CFVs one by one, any CFV may call upon any M_ℓ, and this would have to be loaded from disk. If we overflow RAM, some other M_ℓs will need to be discarded. We can set up a suitable caching protocol to make it more likely that a demanded M_ℓ is found in RAM when needed. Section 3.4 hints that this strategy may not succeed.

The alternative is to invest time up-front to sort the incoming CFVs by key ℓ. The collection of all CFVs sent to a host will usually be large and disk-resident, so actual data (and not just an indirection array of keys) reorganization will be involved in the sort. However, the benefit is that CFVs will now be processed at each host in lemma order. All work for one lemma will be completed before moving on to the next, so only one M_ℓ needs to be in RAM at any time. Thus, our RAM requirement per host will be essentially negligible (beyond the lemma dictionary, usually stored in RAM as a trie).

Summarizing the design discussion up to now,

1) documents are scanned and a sequence of CFVs in no particular ℓ order are emitted from each host,
2) these CFVs are reshuffled through all-to-all communication,
3) all CFVs sent to a destination host are sorted by ℓ,
4) each host loads in sequence a (sub)set of M_ℓs, and completes disambiguation for all CFVs with key ℓ in one chunk.

Compressed corpus size per document	3 KB
Size of CFVs emitted per document	11.8 KB
Time to convert document into CFVs	17 ms/doc
Minimum ambiguity of a lemma	2
Maximum ambiguity of a lemma	742
Minimum number of CFVs for a lemma	1
Maximum number of CFVs for a lemma	23.42 million
Minimum work for a lemma	0.6 ms
Maximum work for a lemma	14h 12m

Figure 7. CFV statistics.

4.2. Preliminary measurements

Figure 7 shows some key statistics about CFVs. Generating CFVs from documents takes about half the time as disambiguating them. However, a 3 KB compressed document blows up to almost four times that size in CFVs. Therefore we also need to estimate the time taken to communicate CFVs across the cluster, and make sure the communication time does not dominate computation time. Our final system sends and receives a total of about 24 GB per host, which (even if not overlapped with computation) takes about 33 minutes in parallel, which is small compared to overall job time.

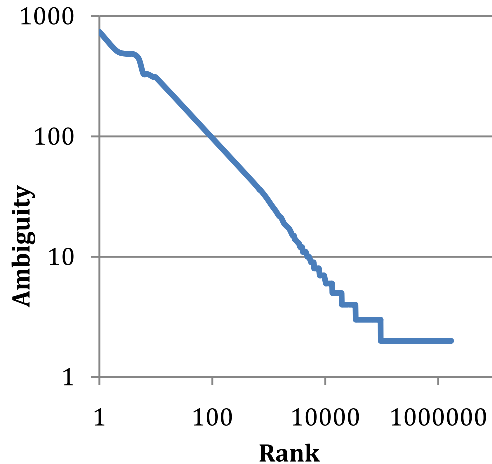

Figure 8. Distribution of ambiguity across lemmas.

Figure 8 shows the distribution of the number of candidate entities ("ambiguity") per lemma (which is highly skewed). Figure 9 shows the distribution of number of CFVs per lemma (which is again highly skewed). The total CPU work for a lemma is the product of the number of CFVs, and the time to process a CFV. We model the latter using the least-square linear regression

$$\text{time/CFV} = 0.0044 \cdot \text{ambiguity} + 0.045 \qquad (1)$$

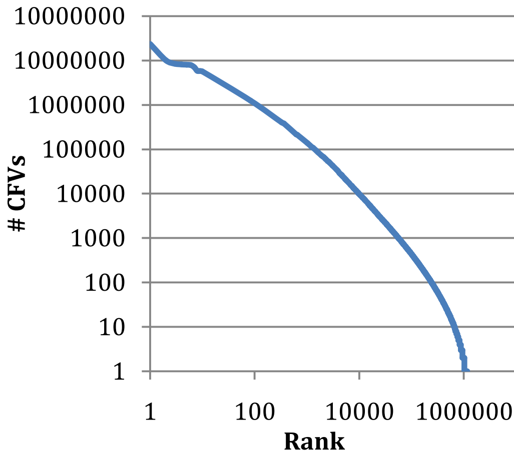

Figure 9. Distribution of CFV corpus occurrence counts across lemmas.

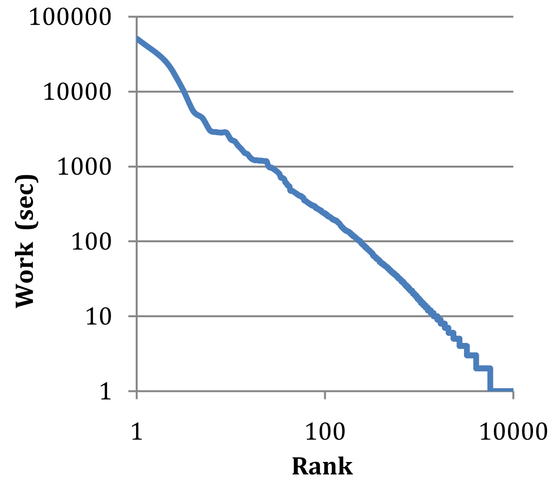

Figure 11. Distribution of total work per lemma.

(see Figure 10). Combining CFVs per lemma with time per CFV, we get the distribution of total work (time) per lemma, shown in Figure 11.

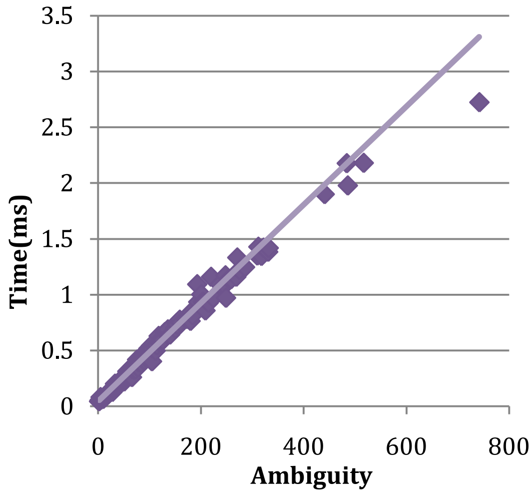

Figure 10. Work per CFV regression.

Notably, being among the heavy hitters (say, top 10) in one dimension (e.g., degree of ambiguity or number of corpus occurrences) is no measure of being a heavy hitter in another dimension (say, total work). This is shown in Figure 12. Among the top 10 lemmas in terms of CFV occurrence count, only 5 appear in the list of top 10 lemmas in terms of work. This highlights the limitation of techniques that attempt to estimate the work in a reduce task based only on the total volume of records destined for the reducer. Also, Figure 10 clearly hints at design of work estimates involving variable per record processing cost for different key-groups as opposed to uniform-cost assumptions [24].

4.3. Greedy CFV allocation and schedule

Even before getting into MR- or Hadoop-specific issues, we collected all CFVs for a lemma into an indivisible task for the lemma, and greedily packed these tasks into 20 hosts,

Degree of ambiguity	Occurrences in corpus	Total time
486	23,424,399	14h 12m
517		3h 12m
332		1h 31m
331		1h 13m
54	10,186,769	48m
71	8,488,375	51m
69	8,126,416	47m
51	8,052,132	
742	7,853,852	7h 13m
29	7,008,439	
47	5,820,516	
42	5,766,605	
8	5,672,619	

Figure 12. Top lemmas in terms of degree of ambiguity, occurrences in the corpus, and total disambiguation time needed. Each row represents a lemma. A blank cell means that lemma was not in the top 10 for that column.

each with 4 cores. (Computation was balanced across 80 cores without regard to communication balance across 20 network interfaces, so this is somewhat naive.) Tasks were sorted in decreasing work order and each successive task sent to the currently least loaded core. While unlikely to be competitive, the advantage of such a schedule is that each lemma model has a "home" host, where its model is loaded exactly once, and all CFVs of that lemma are processed in one batch.

The result is shown in Figure 13. The first host to finish takes 20 minutes, while the last straggler takes 14 hours and 32 minutes. This means that the average host is idle for 13 hours and 41 minutes, or 94% of the time. This is early warning that any MR implementation with a single reduce key per lemma is doomed, and key splitting is vital.

Computation imbalance is accompanied by communication imbalance, as shown in Figure 14. There is little or no clustering of lemmas across the shards of the corpus, so the outflow of CFVs from all hosts is fairly uniform. However, the inflow is highly imbalanced, overloading some hosts with CFVs of

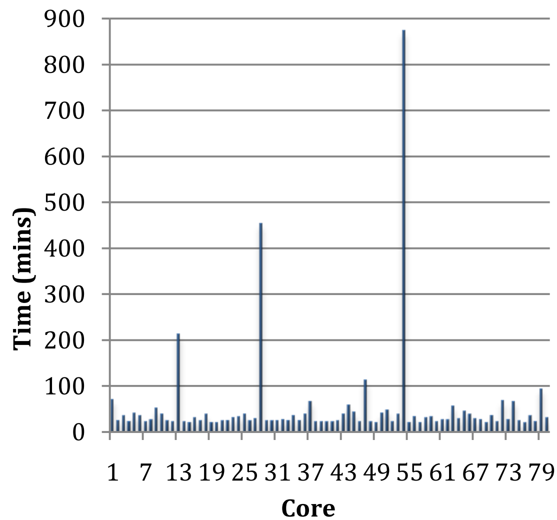

Figure 13. Computation imbalance in greedy scheduling.

"hot" lemmas. Again, note that the number of communication hotspots is different from the number of computation hotspots.

Maximum outgoing traffic/host	11.50 GB
Minimum outgoing traffic/host	8.80 GB
Maximum incoming traffic/host	29.98 GB
Minimum incoming traffic/host	6.83 GB

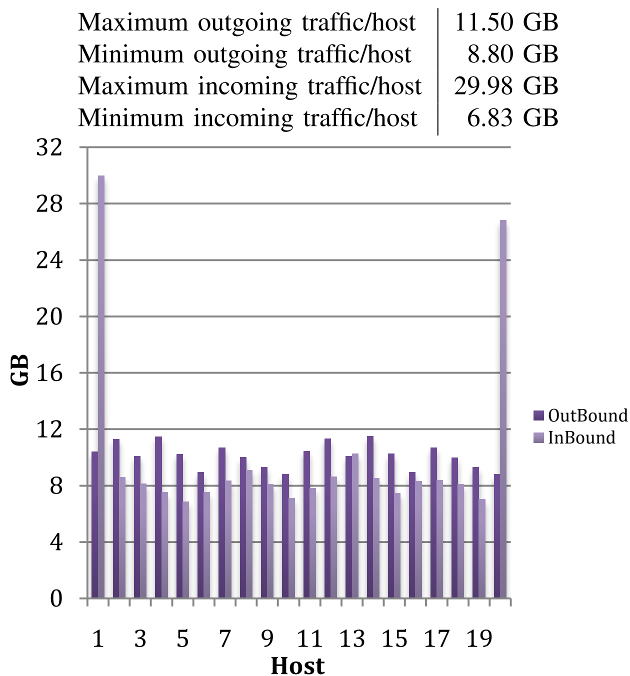

Figure 14. Communication imbalance in greedy scheduling.

5. Using map-reduce

The structure of the computation makes it natural to want to use map-reduce (MR). MR significantly eases coding up certain classes of distributed computations, hiding issues of data layout, storage fault tolerance, communication, and failed jobs. To use MR, we have to implement two interfaces:

$$\text{map}: \quad \text{input} \to \text{list}(k, v)$$
$$\text{reduce}: \quad (k, \text{list}(v)) \to \text{output}$$

Two canonical examples of MR are counting words and preparing an inverted index for a document corpus. To output a count of each word in a corpus of documents, the map task or "mapper" scans and tokenizes input text, and, for each word k, outputs a record $(k, 1)$, with a string and an integer. The "reducer", given input list(k, v), outputs a record $(k, \sum v)$. To prepare the inverted index for a corpus, each document is assigned an integer ID d. While scanning document d, the mapper outputs records (k, d) for each word k. The reducer's input record is list(k, d) and the output is $(k, \text{list}(d))$ where list(d) is a compressed posting list.

5.1. Vanilla Hadoop

In our case, it is most natural to use mappers to transform inputs (text documents) to CFVs keyed by lemma ID, and reducers to transform CFVs to annotation records (outputs). In the most natural use of MR, we would use approximately as many mappers as there are CPU cores, and as many reducers as lemma IDs. Each reducer would load one M_ℓ and handle all CFVs keyed on that lemma ℓ. Thus, a lemma model would never be replicated; all CFVs with that lemma would be sent to the host loading that model.

The first problem with this plan is that no known Hadoop implementation will support tens of thousands to millions of reducers per physical host. Secondly, different lemmas have vastly different amounts of total work, which may lead to substantial reduce load imbalance. Hadoop does allows us to impose external constraints on the maximum number of reducers permitted to coexist on each host (e.g., the number of cores per host). It uses a hash partitioner to allocate keys to reducers. The resulting packing may reduce skew, but, as we shall see, the residual skew will be intolerably large. We call this setup *vanilla Hadoop* hereafter.

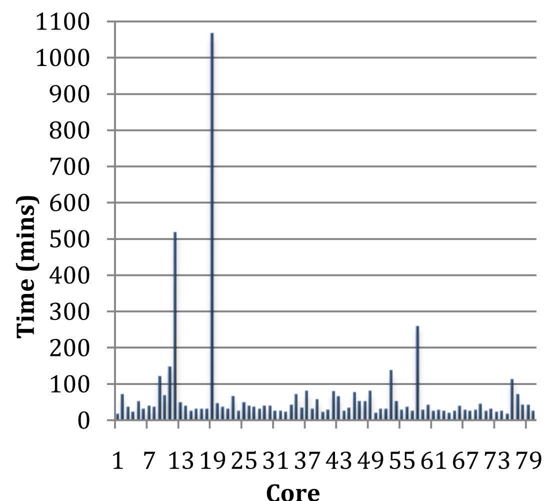

Figure 15. Computation imbalance for vanilla Hadoop.

Figure 15 shows the computation profile of vanilla Hadoop. The time to completion (including communication inside Hadoop) is 20 hours and 19 minutes, but CPUs are idle 82.12% of the time (16 hours, 41 minutes). The shortest reduce takes

16 minutes, while the longest one takes 17 hours and 44 minutes.

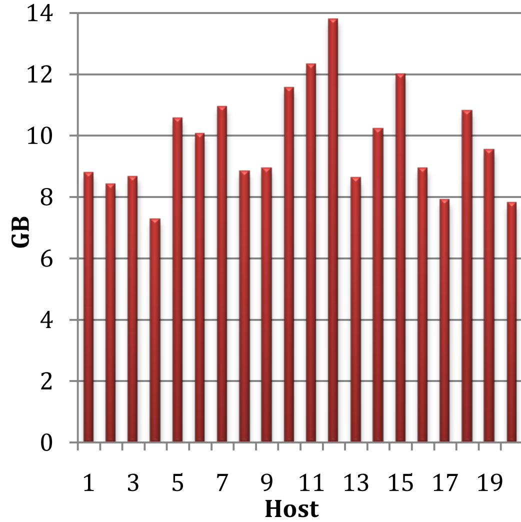

Figure 16. Inbound communication distribution for vanilla Hadoop.

Figure 16 shows the inbound communication profile. The maximum and minimum inbound data volumes are 13.79 GB and 7.28 GB. Note that Hadoop's hash partitioning reduces communication imbalance as compared to Greedy scheduling (Figure 14).

5.2. Skew mitigation by SkewTune

As reviewed in Section 2, many systems have been proposed for skew mitigation. We will evaluate SkewTune [17], one popular recent skew mitigation strategy integrated into Hadoop and available publicly. SkewTune supports dynamic reassignment of reduce tasks to handle user-defined operations (UDOs). SkewTune does not split keys to preserve MR semantics, and this makes it ineffective for our goal. Compared to vanilla Hadoop's 20h 19m, SkewTune takes **19h 31m** to complete, which is only 1.04× faster. CPUs are idle, on an average, for 15h 45m, or 80.7% of the job time.

5.3. Our skew mitigation approach

Technically, we do not really need a standard reducer to perform disambiguation, because, as we have mentioned, M_ℓ can be loaded and used at any number of hosts. Further, all the CFVs with key ℓ need not be combined in any way. (In fact, after disambiguation, they are to be regrouped by *entity ID* e instead, for indexing.)

Given the above discussion, we can set a degree of replication P_ℓ for each lemma model, where $1 \le P_\ell \le P$, P being the total number of reducers we choose to configure. We could select P_ℓ in a variety of ways, based on our offline estimate of total disambiguation CPU work for each ℓ (from Section 4.2). We can then have a fancy scheme to divide the total work for a lemma into its P_ℓ model replicas, but we instead restrict ourselves to the simplest scheme of dividing the work equally amongst the P_ℓ replicas.

Moreover, as we shall see in the next section that an all-or-nothing approach of selecting P_ℓ i.e. setting $P_\ell = P$ for top K jobs and $P_\ell = 1$ for the rest, gives us a solution which lies within a constant factor (much closer to one) of the best possible.

We use the aforementioned heuristic of equal work distribution amongst P_ℓ replicas and limiting P_ℓ to 1 or P on a sample of our dataset and carry out an offline greedy scheduling of resulting tasks (as described in the next section), and finally store the schedule in a file.

We then use the file thus generated to implement the schedule by using a custom partition function (instead of the default hash partitioner) and assign a CFV for a lemma ℓ uniformly at random to one of its P_ℓ disambiguator instances.

Each reducer (disambiguator) processes CFVs in ℓ order, and needs to load and unload each M_ℓ only once. Therefore, our reducers need negligible amounts of RAM for loading models.

5.4. Scheduling objective and approaches

For L independent tasks with task work times $W_1, \ldots, W_\ell, \ldots, W_L$, scheduled on P processors, a lower bound to completion time is

$$\max\left\{\max_\ell W_\ell, \frac{1}{P}\sum_\ell W_\ell\right\}, \tag{2}$$

i.e., the maximum of the largest task time and average task time per processor. A standard (offline) way to address this (NP hard) scheduling problem is to start with P idle processors, sort the tasks in decreasing work times $W_1 \ge \cdots \ge W_L$, and pack the next task to the processor with the currently earliest finish time (EFT). It is easy to see that the EFT schedule finishes all tasks within time

$$\max_\ell W_\ell + \frac{1}{P}\sum_\ell W_\ell, \tag{3}$$

i.e., within a factor of two of the best possible. (In practice, the typical factor is much closer to one.) This is true of tasks that are indivisible. Clearly, $\max_\ell W_\ell$ is the source of trouble. Lin [19] has analyzed this issue in more depth: a typical Zipfian (or power law, or heavy tailed) distribution of task times imposes fundamental limits to parallelism unless task splitting is possible.

As discussed before, suppose M_ℓ is replicated to P_ℓ hosts, and the work W_ℓ is perfectly divisible into these P_ℓ hosts. This creates P_ℓ tasks, each with time M_ℓ/P_ℓ. However, there is a fixed overhead time of c for each resulting task. Then the optimal and EFT schedules are both within a constant factor of

$$\begin{aligned}&\max_\ell\left[\frac{W_\ell}{P_\ell}+c\right]+\frac{1}{P}\sum_\ell P_\ell\left[\frac{W_\ell}{P_\ell}+c\right]\\ &=\max_\ell\left[\frac{W_\ell}{P_\ell}+c\right]+\frac{1}{P}\sum_\ell(W_\ell+cP_\ell).\end{aligned} \tag{4}$$

Therefore the optimization we face is

$$\min_{\{1\le P_\ell \le P: \ell=1,\dots,L\}} \max_\ell \left[\frac{W_\ell}{P_\ell}+c\right]+\frac{1}{P}\sum_\ell (W_\ell + cP_\ell), \quad (5)$$

$$= \underbrace{c+\frac{1}{P}\sum_\ell W_\ell}_{\text{const.}} + \min_{\{1\le P_\ell \le P: \ell=1,\dots,L\}} \max_\ell \frac{W_\ell}{P_\ell} + \frac{c}{P}\sum_\ell P_\ell.$$

where we need to choose replication P_ℓ for each lemma model M_ℓ. This represents P^L combinations.

Instead of searching over those, we will propose candidate values T_0 for the term $\max_\ell W_\ell/P_\ell$, check if each T_0 is feasible, and pick the best overall estimate of finish time T_f over all feasible T_0s.

1: initialize best finish time $T_f \leftarrow \infty$
2: **for** each proposed T_0 **do**
3:   **for** each $\ell = 1,\dots,L$ **do**
4:     let $P_\ell \leftarrow \lceil W_\ell/T_0 \rceil$
5:     **if** $P_\ell > P$ **then**
6:       T_0 is infeasible, continue with next T_0
7:   $T_f \leftarrow \min\{T_f, \max_\ell W_\ell/P_\ell + (c/P)\sum_\ell P_\ell\}$
8: **return** best T_f with corresponding P_ℓs

The following analysis suggests that there is no need to scan through all values of T_0; we will get near-optimal overhead if we check $T_0 = 1, 2, 4, 8, \dots$ for feasibility, then binary-search between the last infeasible and first feasible values for T_0.

In practice, even the binary search for T_0 can be avoided using the following all-or-nothing policy: For some K of the largest jobs, set $P_\ell = P$; for the rest, set $P_\ell = 1$. K is tuned to minimize the above objective. We now give some informal justification why this simpler scheme is good enough for highly skewed task times (as in Figure 11).

Let us model the task time skew using the commonly-used power-law distribution:

$$W_\ell = \frac{T}{\ell^\alpha}, \qquad \text{with } \ell = 1,\dots,L, \quad (6)$$

where $\alpha > 1$ is the power and we have assumed $W_1 \ge \cdots \ge W_L$ without loss of generality. Then the total work in the system is

$$\sum_\ell W_\ell = T\sum_\ell \ell^{-\alpha} \approx T\int_0^L \ell^{-\alpha} d\ell$$

$$= \frac{T}{\alpha-1}\left[1-\frac{1}{L^{\alpha-1}}\right] \approx \frac{T}{\alpha-1} \text{ for large } L. \quad (7)$$

Suppose we split lemmas up to ℓ_0; then, even for the extreme case of $c = 0$, ℓ_0 satisfies

$$\frac{T}{P\ell_0^\alpha} \ge \frac{T}{P(\alpha-1)} = \text{(average work/proc)}, \quad (8)$$

$$\text{or} \quad \ell_0 \le (\alpha-1)^{1/\alpha}. \quad (9)$$

In other words, the same power-law skew that limits parallelism [19] also limits the number of tasks that need to be split for good load balance.

From the previous discussion, we note that when we pick $P_\ell = 1$, the optimal solution has no motivation to allocate $P_\ell > 1$, and so, the excess cost of our heuristic is at most

$$\frac{c}{P}(P-1)\ell_0 \le c(\alpha-1)^{1/\alpha}, \quad (10)$$

or, a *constant number of per-task overheads*. As a sample, $\alpha = 1.2 \implies (\alpha-1)^{1/\alpha} \approx 0.26$, and $\alpha = 3 \implies (\alpha-1)^{1/\alpha} \approx 1.26$.

This approach can be used for any application in general, by optimizing the aforementioned objective to obtain the optimal number of partitions (per key); which can then be used to plan the schedule (greedily), using offline estimates of the work on a sample of data. A custom partition function can then be used to implement the schedule thus obtained.

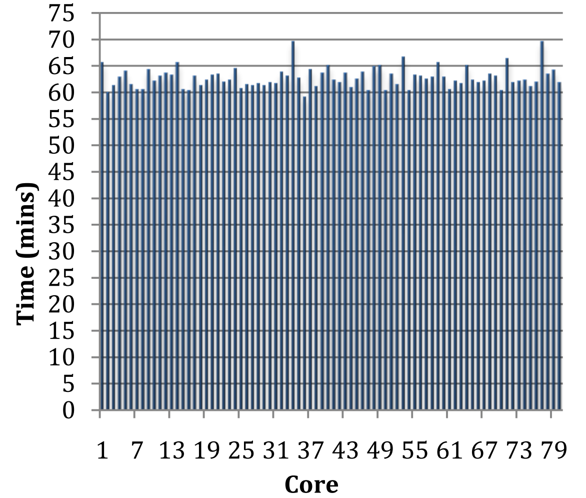

Figure 17. Computation imbalance for our technique.

5.5. *Performance of our approach*

Figure 17 shows the CPU busy times on 80 cores, using our proposed scheduler. The overall job time reduces from 19h 31m to **3h 47m** with an additional (one-time) overhead of 50 minutes for creation of schedule using a sample 6 GB corpus. The average CPU idle time is 7 minutes, or 3% of job time. The maximum and minimum reducer times are 69 and 59 minutes, representing excellent load balance. Figure 18 shows that inbound communication is also well-balanced, although no conscious effort was made in that direction: the maximum and minimum number of bytes destined to a host were 10.7 GB and 8.65 GB.

6. Conclusion

We have described the evolution of a critical Web annotation application from a custom implementation to a highly optimized version based on Hadoop MR. The evolution was critical as an essential data structure began to exceed RAM size on our cluster. We started with two incremental approaches, but their performance was unacceptable. Then we attempted to use standard Hadoop, but hit a serious (reduce) skew problem, which seems endemic in MR applications in this domain. We

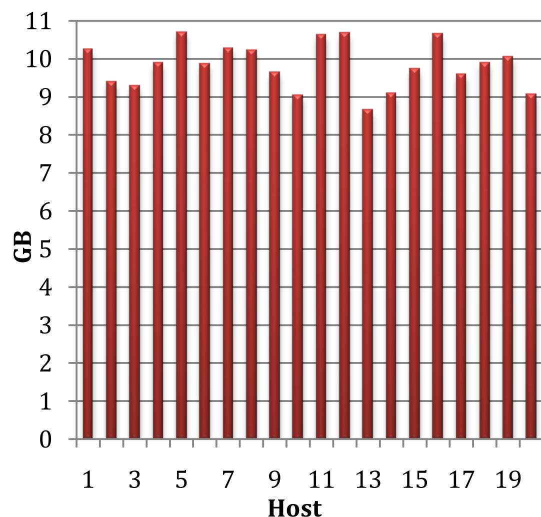

Figure 18. Communication imbalance for our technique.

also tried a recent skew mitigation strategy (SkewTune) but with only modest improvement. Inspired by recent work in reduce key-splitting, we finally designed our own methods for load estimation, key-splitting, and scheduling.

A precise performance model was first created, which led to an objective function that we optimized heuristically. This approach led to large benefits: our final annotator was 5.4× faster than standard Hadoop MR, and 5.2× faster than even SkewTune. Our technique and tweaking on top of Hadoop may be of independent interest, and may be desirable to add to the Hadoop library as supplementary part of the MR API.

If we could materialize all CFVs to disk, we might express disambiguation as a distributed equi-join (on lemma ID), using HBase or Pig [22], and then write user-defined functions on joined tuples to complete the disambiguation. However, we measured the volume of CFVs emitted per document to be almost four times its compressed size. For our 1.6 TB corpus, that means 6.77 TB of CFVs stored in HDFS (which will further impose a 3× replication). If one can afford that kind of storage for transient data, it may be worthwhile exploring the join option.

Acknowledgment: Thanks to Siddhartha Nandi, Ameya Usgaonkar, and Atish Kathpal for many helpful discussions. The work was partly supported by a grant from NetApp.

References

[1] S. Chakrabarti. Bridging the structured-unstructured gap: Searching the annotated Web. Keynote talk at WSDM 2010, Feb. 2010.

[2] S. Chakrabarti, S. Kasturi, B. Balakrishnan, G. Ramakrishnan, and R. Saraf. Compressed data structures for annotated web search. In *WWW Conference*, pages 121–130, 2012.

[3] S. Chakrabarti, D. Sane, and G. Ramakrishnan. Web-scale entity-relation search architecture (poster). In *WWW Conference*, pages 21–22, 2011.

[4] ClueWeb12. http://lemurproject.org/clueweb12/.

[5] CSAW. http://www.cse.iitb.ac.in/~soumen/doc/CSAW/.

[6] S. Cucerzan. Large-scale named entity disambiguation based on Wikipedia data. In *EMNLP Conference*, pages 708–716, 2007.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, Dec. 2004.

[8] Freebase. http://www.freebase.com/.

[9] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Handling data skew in MapReduce. In *International Conference on Cloud Computing and Services Science*, pages 100–109, Noordwijkerhout, The Netherlands, 2011.

[10] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in MapReduce based on scalable cardinality estimates. In *ICDE*, pages 522–533, 2012.

[11] J. Guo, G. Xu, X. Cheng, and H. Li. Named entity recognition in query. In *SIGIR Conference*, pages 267–274. ACM, 2009.

[12] Hadoop. http://hadoop.apache.org/.

[13] HBase. http://hbase.apache.org/.

[14] J. Hoffart et al. Robust disambiguation of named entities in text. In *EMNLP Conference*, pages 782–792, Edinburgh, Scotland, UK, July 2011. SIGDAT.

[15] S. Kulkarni, A. Singh, G. Ramakrishnan, and S. Chakrabarti. Collective annotation of Wikipedia entities in Web text. In *SIGKDD Conference*, pages 457–466, 2009.

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 75–86, Indianapolis, Indiana, 2010. ACM.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: mitigating skew in MapReduce applications. In *SIGMOD Conference*, pages 25–36, Scottsdale, Arizona, 2012. ACM.

[18] Y. Kwon, K. Ren, M. Balazinska, and B. Howe. Managing skew in hadoop. *IEEE Data Engineering Bulletin*, 36(1):24–33, 2013.

[19] J. Lin. The curse of Zipf and limits to parallelization: A look at the stragglers problem in MapReduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.

[20] Memcached. http://memcached.org/.

[21] R. Mihalcea and A. Csomai. Wikify!: linking documents to encyclopedic knowledge. In *CIKM*, pages 233–242, 2007.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, Vancouver, Canada, 2008. ACM.

[23] P. Pantel, T. Lin, and M. Gamon. Mining entity types from query logs via user intent modeling. In *ACL Conference*, pages 563–571, Jeju Island, Korea, July 2012.

[24] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in MapReduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 16:1–16:14, San Jose, California, 2012.

[25] K. Ren, G. Gibson, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence; a comparative workloads analysis from three research clusters. In *SC Companion (Posters)*, page 1453, 2012.

[26] U. Sawant and S. Chakrabarti. Learning joint query interpretation and response ranking. In *WWW Conference*, Brazil, 2013.

[27] F. M. Suchanek, G. Kasneci, and G. Weikum. YAGO: A core of semantic knowledge unifying WordNet and Wikipedia. In *WWW Conference*, pages 697–706. ACM Press, 2007.