Low-Latency, High-Throughput Access to Static Global Resources within the Hadoop Framework

Jimmy Lin,[1] Anand Bahety,[2] Shravya Konda,[2] and Samantha Mahindrakar[1]
[1]The iSchool, College of Information Studies
[2]Department of Computer Science
University of Maryland
College Park, Maryland 20742
{jimmylin,abahety,shravyak,sam}@umd.edu

HCIL Technical Report HCIL-2009-01
(Version of January 5, 2009)

Abstract

Hadoop is an open source implementation of Google's MapReduce programming model that has recently gained popularity as a practical approach to distributed information processing. This work explores the use of memcached, an open-source distributed in-memory object caching system, to provide low-latency, high-throughput access to static global resources in Hadoop. Such a capability is essential to a large class of MapReduce algorithms that require, for example, querying language model probabilities, accessing model parameters in iterative algorithms, or performing joins across relational datasets. Experimental results on a simple demonstration application illustrate that memcached provides a feasible general-purpose solution for rapidly accessing global key-value pairs from within Hadoop programs. Our proposed architecture exhibits the desirable scaling characteristic of linear increase in throughput with respect to cluster size. To our knowledge, this application of memcached in Hadoop is novel. Although considerable opportunities for increased performance remain, this work enables implementation of algorithms that do not have satisfactory solutions at scale today.

1 Introduction

Hadoop is an open source implementation of Google's MapReduce framework [7] for distributed processing. It has recently gained popularity as a practical, cost-effective approach to tackling data-intensive problems, and has stimulated a flurry of research on applications to machine learning [6, 28], statistical machine translation [10], corpus analysis [19], and text retrieval [11]. Additionally, researchers have examined extensions to the programming model [29], developed a complementary dataflow language [23], and also built alternative implementations on different architectures [14, 16, 26]. It is no exaggeration to say that MapReduce in general, and Hadoop in particular, has reinvigorated work on applications, algorithms, and approaches to distributed processing of large datasets.

The MapReduce programming model,[1] which can be best described as a functional abstraction, can succinctly capture a large class of distributed algorithms (see Section 2). For these problems, Hadoop provides scalable and robust solutions that can be implemented by any researcher. Nevertheless, there are large classes of problems for which Hadoop presently provides no satisfactory solution.

[1] To be precise, in this paper we use MapReduce to refer to the programming model and use Hadoop to refer to the specific open-source implementation.

In this paper, we tackle a class of problems which can be expressed in MapReduce, but cannot be practically executed at scale due to limitations of the Hadoop framework and the design decisions it embodies. These problems all share in the need for low-latency, high-throughput access to static global resources "on the side" while processing large amounts of data. Hadoop does not have a mechanism for low-latency random access to data stored on disk, and therefore is unable to efficiently accommodate algorithms that require this capability. We propose a solution to this problem by a novel use of memcached, a open-source, high-performance, distributed in-memory object caching system widely used in database-driven Web applications. The primary contribution of this work lies in an architectural enhancement to Hadoop that enable solutions to a large class of algorithms previously impractical at scale. Use of an existing commercial-grade, off-the-shelf tool to achieve this capability means that our solution can be rapidly deployed in both research and production environments.

We begin with an overview of the MapReduce programming model (Section 2) and a number of motivating real-world problems (Section 3). Our proposed solution consists of a distributed in-memory caching system implemented by memcached (Section 4). This is evaluated with a simple demonstration application of scoring natural language sentences with a unigram language model (Section 5). Experimental results illustrate that memcached provides a feasible general-purpose solution for rapidly accessing global key-value pairs from within Hadoop programs (Section 6). Our proposed solution exhibits the desirable characteristic of linear increase in throughput with respect to cluster size. The initial implementation appears to be sufficient for the statistical machine translation applications that motivated this work. Nevertheless, we identify considerable opportunities for performance improvements and discuss tradeoffs involved in the architecture (Section 7) before concluding (Section 8).

2 Background

The only practical solution to large-data challenges today is to distribute computations across multiple machines in a cluster. With traditional parallel programming models (e.g., MPI), the developer shoulders the burden of explicitly managing concurrency. As a result, a significant amount of the programmer's attention is devoted to system-level details such as interprocess communication, lock management, and error handling. Recently, MapReduce [7] has emerged as an attractive alternative; its functional abstraction provides an easy-to-understand model for designing scalable, distributed algorithms.

2.1 The Programming Model

MapReduce builds on the observation that many information processing tasks have the same basic structure: a computation is applied over a large number of records (e.g., Web pages or nodes in a graph) to generate partial results, which are then aggregated in some fashion. Naturally, the per-record computation and aggregation function vary according to task, but the basic structure remains fixed. Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction for programmer-defined "mappers" and "reducers", which have the following signatures:

map: $(k_1, v_1) \rightarrow [(k_2, v_2)]$
reduce: $(k_2, [v_2]) \rightarrow [(k_3, v_3)]$

Key-value pairs form the processing primitives in MapReduce. The mapper is applied to every input key-value pair to generate an arbitrary number of intermediate key-value pairs (the notation $[\ldots]$ is used to represent a list). The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs. This two-stage processing structure is illustrated in Figure 1. Typically, an application would consist of multiple MapReduce jobs chained together. For example, in

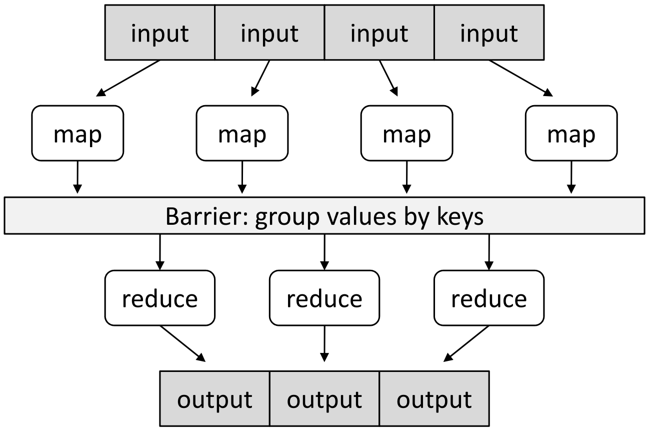

Figure 1: Illustration of the MapReduce framework: the "mapper" is applied to all input records, which generates results that are aggregated by the "reducer". The runtime groups together values by keys.

the standard MapReduce implementation of the well-known PageRank [24] algorithm, each iteration corresponds to a MapReduce job.

Under this framework, a programmer needs only to provide implementations of the mapper and reducer. On top of a distributed file system, the runtime transparently handles all other aspects of execution on clusters ranging from a few to a few thousand cores. The runtime is responsible for scheduling map and reduce workers on commodity hardware assumed to be unreliable, and thus is tolerant to various faults through a number of error recovery mechanisms. In the distributed file system, data blocks are stored on the local disks of cluster machines; the MapReduce runtime attempts to schedule mappers on machines where the necessary data resides, thus "moving the code to the data". It also manages the potentially very large sorting problem between the map and reduce phases whereby intermediate key-value pairs must be grouped by key.

Google's proprietary implementation of MapReduce is in C++ and built on top of the Google File System (GFS) [13]; it is not available to the public. However, Hadoop, an open-source Java implementation led by Yahoo, allows any developer take advantage of MapReduce. HDFS is an open-source implementation of GFS that provides the storage substrate underlying Hadoop.

2.2 Design Tradeoffs and Limitations

The performance characteristics of MapReduce implementations are directly tied to assumptions built into the underlying distributed file systems—GFS in the case of Google. Since HDFS is an implementation of GFS for Hadoop, it inherits the same set of assumptions. GFS was optimized for sequential reads and appends on large files, and therefore emphasizes high sustained throughput over low latency. The file system is deployed as a simple master-slave architecture: the master stores metadata and namespace mappings to data blocks, which are themselves stored on the slaves (called chunkservers in GFS or datanodes in HDFS). The master only communicates metadata; all data transfers are coordinated directly with the slaves. Overall, MapReduce is primarily designed for batch-oriented processing—computations that involve sequential reads of large files.

In the present implementation of Hadoop, latency for reading a random chunk of data from HDFS can range from tens of milliseconds in the best case to possibly hundreds of milliseconds in the worst case. In the best case scenario, the location of the data block is already known (e.g., from a previous request) and the data block is stored locally. In this situation, the only latency involved is the disk seek (assuming the data hasn't been cached), data transfer, and the overhead of the file system itself (which runs in user space on top of a standard Linux file system). In the worst case scenario, the location of the data block is not known and the data block is stored on another machine in the cluster. In this situation, the client must first negotiate with the master to find the location of the desired data block, contact the correct datanode (slave), and then initiate transfer of the data over the network. To make matters worse, latencies are mostly unpredictable, as the file system API tries to hide the physical location of the data blocks from the client (for robustness reasons).

The upshot is that HDFS does not provide a practical mechanism for low-latency random access to data stored on disk. The open-source community has yet to figure out a good solution to this problem. One evolving practice is to perform batch processing in Hadoop, and then copy the results out of HDFS into to a more mature platform for real-time applications, e.g., the traditional Linux, Apache, MySQL, PHP (so-called LAMP) software stack. This is far from an ideal situation, since it may require storing two separate copies of the data and dealing with consistency issues.

3 The Problem

This paper tackles the problem of MapReduce algorithms in Hadoop that require low-latency, high-throughput access to static global resources "on the side" while processing large amounts of data. Although a large number of algorithms have this characteristic, we were specifically motivated by three real-world problems discussed below, two of which arise from our colleagues' work on statistical machine translation. As we explain, Hadoop presently provides no satisfactory solution to these problems, which fundamentally derive from the design choices discussed in Section 2.2.

Language Model Probabilities. Many natural language processing algorithms require access to *n*-gram probabilities from a language model. An *n*-gram language model defines a probability distribution over sequences of words, called *n*-grams, usually estimated from a large number of observations (i.e., a large text collection); see [21] for an introduction. One concrete example is the decoding (i.e., translation) phase in a statistical machine translation system [15, 20], where translation hypotheses need to be scored against a language model (to maximize the probability that the generated output is fluent natural language, subjected to various translation constraints). One popular optimization technique, called minimum error-rate training (MERT) [22, 27], requires the repeated decoding of many training sentences, which can be straightforwardly parallelized with MapReduce using mappers. As the algorithm maps over training sentences, it needs to rapidly query the language model to retrieve relevant probabilities. This problem is trivial if the language model is manageable in size—each mapper simply loads all model probabilities into memory. The fundamental bottleneck here of course is memory, which constrains the maximum size of the language model. Web-scale language models can have billions of parameters [2] and may not easily fit into memory on a single machine. Client-server architectures for providing access to distributed language models exist [31, 12], but in Section 4 we propose a more general-purpose solution.

Model Parameters in Iterative Algorithms. Expectation-maximization, or EM [9], describes a class of algorithms that iteratively estimates parameters of a probabilistic model with latent (i.e., hidden) variables, given a set of observations. Each iteration requires access to model parameters from the previous iteration, thus presenting a distributed data access problem. One well-known EM algorithm is word alignment in statistical machine translation [3], where the task is to induce word translation probabilities based on parallel training data (pairs of sentences in different languages known to be translations of each other). Word alignment is a critical step in phrase-based statistical machine translation [15]. Dyer et al. [10] have previously described MapReduce implementations of two well-known algorithms. Model parameters are loaded into memory at the beginning of each iteration, and execution of the algorithm yields updated model parameters, which are written to the distributed file system in a compressed encoding, ready for the next iteration. Of course, this approach assumes that model parameters are small enough to fit into memory, which presents a scalability bottleneck.[2]

Joins Across Relational Datasets. Although there are multiple ways to perform joins across relational datasets in MapReduce, one of the most efficient approaches is to load one of the datasets into memory, and map over the other dataset, looking up records from the first by the relevant join key. To give a concrete example, suppose one had a Web log of user clickthroughs (with unique user ids) and user demographics (keyed by the user id) stored separately. Aggregation of clicks by demographic characteristics would require a join across these two datasets. This approach is fundamentally limited by the size of one of the datasets.

For the above use cases, one might consider partitioning the global resource into chunks, where each chunk contains only the data that is required by a particular mapper or reducer. For example, the translation of any individual sentence requires only a handful of n-gram probabilities. The difficulty, however, lies in predicting *which ones*. During the decoding process in machine translation, the system does not know which hypotheses need to be considered (and therefore which probabilities are required) until the algorithm is well underway. Two-pass solutions are possible, but at the cost of added complexity.

A distributed data store (e.g., Google's Bigtable [4] or Amazon's Dynamo [8]) might provide a possible solution for accessing global resources that are too big to fit into memory on any single machine (and cannot be easily partitioned). HBase[3] is an open-source implementation of BigTable, but based on our experiments, it does not appear to be fast enough to handle the demands of the above applications. Moreover, it is not entirely clear if such systems are cost effective, in the sense that database functionality is not required for many applications, and a lighter-weight solution might suffice. We propose exactly such a solution.

4 Proposed Solution

For the reasons outlined in Section 2.2, Hadoop lacks a mechanism for providing low-latency access to data in HDFS. To address this limitation in the context of the problems described in Section 3, we need a distributed key-value store with the following characteristics:

- a uniform global namespace;
- low-latency, high-throughput access to large numbers of key-value pairs;
- a scalable architecture.

In-memory caching of on-disk data is a sensible solution to meeting these requirements. In fact, we realized that an existing open-source tool can be adapted to provide these capabilities in the Hadoop environment.

Memcached[4] (pronounced mem-cash-dee) is a high-performance, distributed in-memory object caching system. Although application-neutral in design, it was originally developed to speed up database-driven Web applications by caching the results of database queries. This open-source tool is currently used by a number of popular Web sites, including Live Journal, Slashdot, Wikipedia, Facebook, among others.[5] A simple database-driven Web architecture that uses memcached is shown

[2] See [28] for an alternative approach to distributed EM with custom solutions based on different network typologies.

[3] http://hadoop.apache.org/hbase/

[4] http://www.danga.com/memcached/

[5] http://www.danga.com/memcached/users.bml

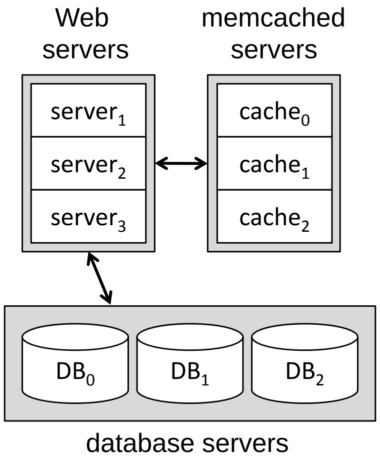

Figure 2: Typical use of memcached to accelerate database-driven Web applications by caching the result of database queries.

in Figure 2. Typically, memcached is initialized as daemon processes on a cluster of machines and provides a global key-value store through a simple interface. Each server is responsible for a portion of the key space and acts independently of the others in a share-nothing architecture. The daemon itself is highly optimized and scales to a large number of open network connections. The client API maintains a list of available memcached servers and is responsible for hashing keys to "buckets" and assigning each to a particular server.

This work explores the use of memcached to provide low-latency, high-throughput access to static global resources in Hadoop. Use of memcached proceeds in two steps:

- First, a global resource, encoded as key-value pairs, is loaded into memcached from HDFS. Populating the cache can be parallelized, expressed as a MapReduce job with no reduce phase. This is illustrated in Figure 3 (top).
- Once the global resource has been loaded, subsequent MapReduce programs can use the client API to access key-value pairs in the global resource. Each mapper or reducer maintains connections to the memcached servers, and therefore all requests happen in parallel and are distributed across all memcached servers. This is illustrated in Figure 3 (bottom).

Execution of an algorithm might, for example, involve multiple cache loads and subsequent processing steps. For example, each iteration of distributed EM might involve loading model parameters into memcached and then running a MapReduce algorithm over the training data, which writes updated model parameters to disk. The new parameters would then be loaded into memcached for the next iteration, repeating until convergence.

We have implemented the capabilities described here in *Cloud⁹*, an open-source MapReduce library for Hadoop,[6] being developed at the University of Maryland for teaching [18] and a number of ongoing research projects [11, 19]. Although the idea of providing distributed access to key-value pairs using a client-server architecture is certainly not new (and researchers have implemented specialized applications such as distributed language models [31, 12, 2]), to our knowledge, this use of memcached within Hadoop is novel and represents a general purpose solution.[7] Of course, Google is likely to have developed similar mechanisms for addressing the same class of problems, but we are not aware of any

[6] http://www.umiacs.umd.edu/∼jimmylin

[7] Recently, our colleague Chris Dyer (personal communication) has independently deployed memcached as a language model server, but on a single large-memory machine outside the Hadoop environment.

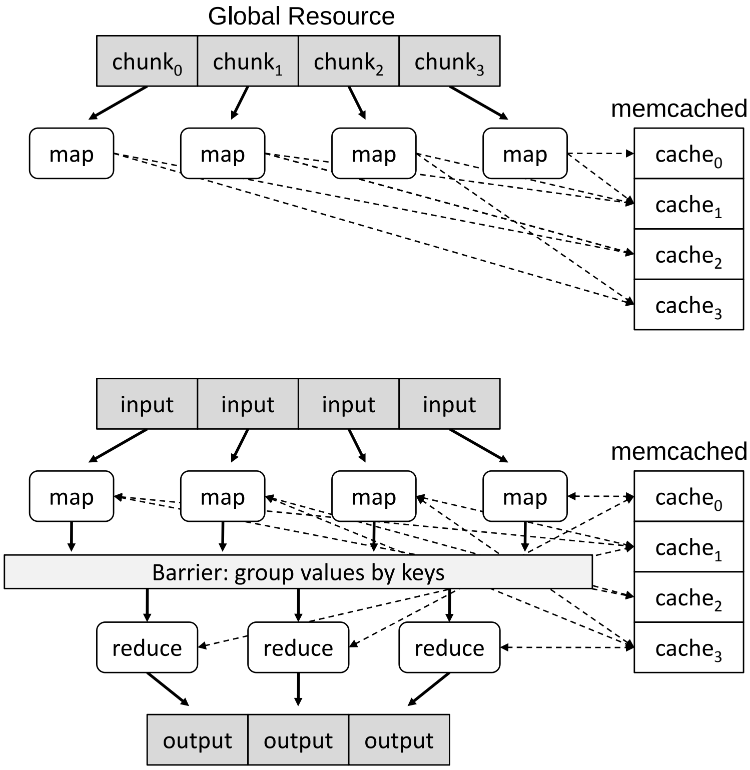

Figure 3: Diagrams illustrating the integration of memcached with Hadoop. Top diagram shows loading a static global resource (key-value pairs) into memcached servers in parallel (as a MapReduce job with no reducers). Bottom diagram illustrates a MapReduce program accessing cached key-value pairs from within mappers and reducers.

publications describing their solution (but solutions are hinted at in [4, 2]). Nevertheless, we are able to provide this capability with an existing commercial-grade, open-source tool, which means that our solution can be rapidly deployed in both research and production environments.

5 Methods

We devised a series of experiments to determine whether memcached can provide low-latency, high-throughput access to static global resources within the Hadoop framework. Our demonstration application implements the simple task of scoring a large collection of natural language sentences given a unigram language model. Although this exact task is of little practical use, it captures the general structure of natural language processing algorithms that require access to language model probabilities (see Section 3). As the primary contribution of this paper is an architectural enhancement to Hadoop that enables solutions to large classes of problems, we stake our claim on performance characteristics of our proposed caching mechanism.

5.1 Demonstration Application

A unigram language model encodes a “bag of words” assumption, in which text is treated as unordered, independent collections of terms (i.e., a multinomial model). Although such a formulation ignores the richness and complexity of natural language—disregarding syntax, semantics, and even word order—this simple model has proven to be effective in text retrieval applications [25, 17].

```
procedure MAP($k, s$)
    $p \leftarrow 0$
    for all $w \in s$ do
        $p \leftarrow p + \text{LOGP}(w)$
    EMIT($k, p$)
```

Figure 4: Pseudo-code of algorithm for scoring a sentence with a unigram language model.

Given a sequence of natural language words (i.e., a sentence), we can compute the probability that a unigram language model M will "generate" the word sequence as follows:

$$p(s|M) = \prod_{w \in s} p(w|M) \tag{1}$$

Working with log probabilities is preferable, since it allows us to convert the product into a sum:

$$\log p(s|M) = \sum_{w \in s} \log p(w|M) \tag{2}$$

For sentences of equal length, this quantity allows us to determine which is more likely to be observed according to the language model. More sophisticated language models are used in qualitatively similar ways to score the "fluency" of natural language output (e.g., scoring machine translation hypotheses).

In our experiments, we built a unigram language model based on relative counts (also known as the maximum likelihood estimate):

$$\log p(w|M) = \log \frac{c(w)}{\sum_w c(w)} \tag{3}$$

where $c(w)$ represents the count of word w in the entire collection. It is well-known that smoothing, the process of assigning non-zero probabilities to unseen events, plays an important role in language modeling [21, 30]. For the purposes of characterizing cache performance, however, the exact probability distribution is unimportant.

Computing the maximum-likelihood log probabilities is easily expressed as two MapReduce jobs—the first performs a simple word count (a well-known algorithm; see [7]), and the second converts these raw counts in log probabilities. An HDFS file containing (word, log probability) pairs comprise the static global resource in our experiments.

With log probabilities loaded into memcached, scoring a large collection of sentences is quite straightforward. The pseudo-code of this algorithm is shown in Figure 4. The algorithm maps over all input sentences (each identified by a unique id k); each mapper tokenizes the input sentence s and queries memcached for the log probability of each word w, shown by the function LOGP$(\cdot)$. The sum of log probabilities is emitted as the output. There is no need for a reducer in this algorithm.

We stress that this simple demonstration application is not intended to be useful in its own right, but rather was designed to probe the performance characteristics of the distributed cache. Therefore, we intentionally avoided performing any significant computation within the mapper so that we can obtain accurate measurements of raw query throughput.

5.2 Experimental Setup

For the demonstration application, we used monolingual English newswire training data prepared for Fourth Workshop on Statistical Machine Translation.[8] This corpus contains approximately 21.2m

[8] http://www.statmt.org/wmt09/

English sentences totaling 2.6 GB (uncompressed). Given that statistical machine translation was one motivating application, we felt that using "real data" was appropriate—this collection was among those provided to participants in a shared translation task organized as part of the workshop. We adopted the naïve tokenization strategy of simply breaking on whitespace and discarding all tokens longer than 100 characters (garbled input); this yielded a total of 438.2m tokens (we avoid calling them "words" since many of them are not). The entire collection contains 2.75m unique tokens.

All experiments were conducted on Amazon's Elastic Compute Cloud (EC2), a "cloud computing" service that "rents" virtualized clusters. The basic unit of resource in EC2 is an "instance-hour"; specifications for different instance types (as well as cost) vary. Each small instance used in our experiments is roughly comparable to one processor core, with 1.7 GB of memory. In the context of our experiments, EC2 provided a convenient way to provision clusters of different sizes. We performed experiments on clusters running Hadoop version 0.17.0, with 20, 40, 60, and 80 slave instances (plus an additional instance for the master). We used memcached (version 1.2.6) and the spymemcached Java client API (version 2.2)[9] to query the language model probabilities. For each cluster configuration, memcached was started on all EC2 instances (including the master), with 1 GB RAM allocated for object storage. In each experimental run, the number of mappers was set to twice the number of slave instances in the cluster (in the default parameter settings of Hadoop, each worker can run two mappers concurrently). Speculative execution was disable. The number of reducers was set to zero, so all output was directly written to HDFS.

Since the purpose of these experiments was to characterize the performance of memcached in Hadoop, the two figures of merit were latency and throughput. Latency for each memcached query was measured by computing the difference in system time (measured in milliseconds) before and after the memcached client API call (i.e., the LogP function in Figure 4). The cumulative latency was recorded via a custom counter in Hadoop. Counters in Hadoop provide a lightweight mechanism for reporting simple statistics, such as the number of records processed. The framework allows programmer-defined counter types. Average latency per memcached query for the entire run was determined by dividing the cumulative latency by the total number of queries (which is equal to the total number of tokens in the entire text collection, or 438.2m in our case).

Throughput was computed in a different manner. Since we are interested in the overall throughput of the entire system, this figure cannot be derived from the cumulative latency since the queries happen in parallel. Of course, we could measure the end-to-end running time of the entire Hadoop job. This measurement, however, would include the overhead of the Hadoop runtime itself (e.g., initialization of Java Virtual Machines, reading data from and writing results to HDFS, etc.). To account for this overhead, we performed another run that was exactly the same as the algorithm in Figure 4 except for the LogP memcached query. Instead of querying the cache, the algorithm simply summed a number of constants equal to the number of tokens in the sentence. This "NoOp" variant, which was run on the same exact dataset, allowed us to quantify the overhead of the Hadoop framework itself. To compute throughput, we subtracted the end-to-end running time of the "NoOp" variant from the memcached run and divided the result by the total number of memcached queries.

6 Results

Results of our experiments are shown in Table 1. With an 80-slave EC2 cluster, the demonstration application was able to sustain memcached throughput of 74.2k queries per second at an average access latency of 2.05 ms; the entire job of computing language model scores on 21.2m sentences, totaling 438.2m tokens, completed in 1h 39m. It appears that the Hadoop runtime overhead is negligible compared to the time required to query the memcached servers, as shown by the running times of the

[9] http://code.google.com/p/spymemcached/

Instances	**Running Time**	**$\sum$ Latency**	**Avg. Latency**	**"NoOp"**	**Throughput**
20	5h 27m	7.43×10^8 ms	1.69 ms	84s	2.24×10^4 q/s
40	3h 06m	8.59×10^8 ms	1.96 ms	44s	3.94×10^4 q/s
60	2h 09m	9.04×10^8 ms	2.06 ms	43s	5.66×10^4 q/s
80	1h 39m	8.97×10^8 ms	2.05 ms	41s	7.42×10^4 q/s

Table 1: Results of scoring natural language sentences with a unigram language model in our proposed Hadoop memcached architecture. Columns show: number of EC2 instances, end-to-end running time, cumulative latency, average latency per query, running time of the "NoOp" variant, and query throughput.

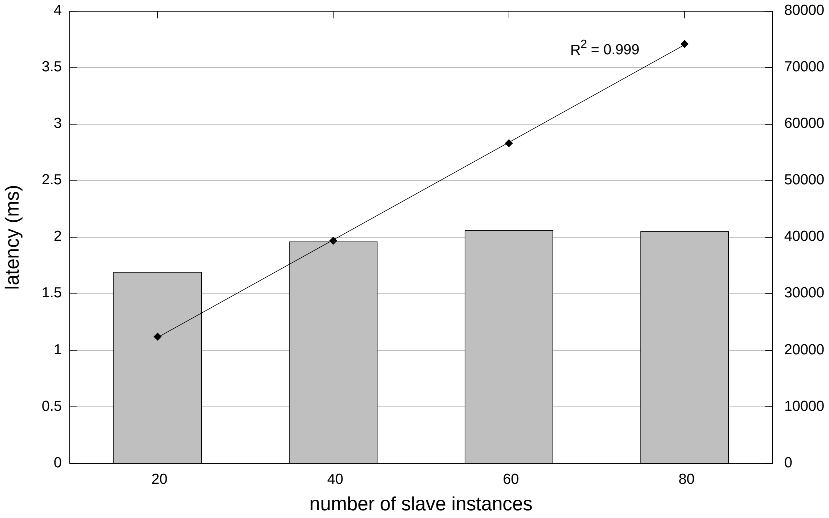

Figure 5: Graph showing latency (left y axis, bar graph) and throughput (right y axis, line graph) for various Hadoop memcached cluster configurations.

"NoOp" variants. Thus, for the purposes of computing throughput, it makes little difference whether or not the costs of the runtime are taken into account.

Figure 5 displays latency and throughput figures in graphical format to convey the scaling characteristics of the architecture with respect to size of cluster. Latency (shown as the bar graph) for the 20-slave cluster appears to be lower, but latency for the other cluster configurations appear to be about the same. Throughput (shown as the line graph) increases linearly with the the size of the cluster. A linear regression line yields a nearly perfect fit ($R^2 = 0.999$).

The distribution of key-value pairs (i.e., tokens and associated log probabilities) in the 80-instance cluster is shown in Figure 6. The default partitioning scheme in the spymemcached Java client API does a good job distributing the tokens across the memcached server instances.

7 Discussion

Memcached is known to be highly effective in speeding up database-driven Web applications by caching the results of database queries. We have shown that it can also be adapted to provide low-latency, high-throughput access to static global resources in Hadoop. Our current "out of the box" deployment achieves a level of performance that is sufficient for our motivating statistical machine translation applications. Translation involves a processor-intensive search through the hypothesis space, which

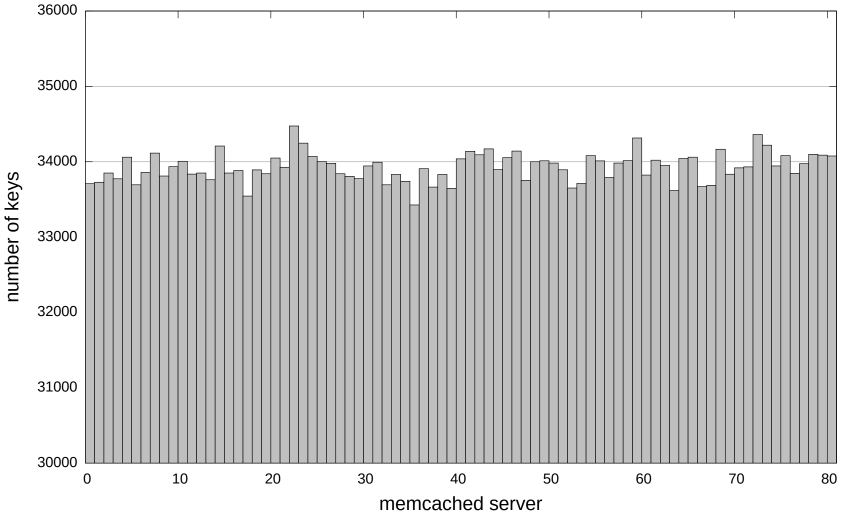

Figure 6: Distribution of key-value pairs (i.e., tokens and associated log probabilities) across memcached servers in the 80-slave cluster configuration.

can take a few seconds with standard phrase-based models [15] and significantly longer (more than ten seconds) for more sophisticated hierarchical models [5].

7.1 Further Optimizations

Nevertheless, there are considerable opportunities for further increasing cache performance. These represent fairly standard engineering practices, but are nevertheless worth discussing. First, it is important to note that results in Section 6 were obtained from experiments on EC2, which is a virtualized environment where processes incur a noticeable performance overhead (see, for example, [1]). While this affects all cluster configurations equally and will not likely impact results qualitatively, absolute numbers may not be entirely accurate.

In the current setup, latency appears to be the limiting factor on performance. In the 80-slave configuration, there are 160 mappers simultaneously querying memcached; a latency of 2.05ms translates into an upper bound on throughput of 7.8×10^4 query per second. This is not much higher than the actual observed throughput, suggesting that the mappers spend most of the time waiting for responses from the memcached servers (since the demonstration application runs synchronously). Note that because cache queries involve the network, round-trip communication time between any pairs of machines places a lower bound on latency. In a typical hosted environment, this would be on the order of a few tenths of a millisecond.

Even with this lower bound on latency, there are some straightforward ways to significantly increase throughput with client-side optimizations. Synchronous memcached queries could be made asynchronous—but this comes at the cost of additional complexity in handling concurrency on the client side (inside each mapper), which is perhaps counter to the spirit of MapReduce. An easier optimization is to make cache requests in larger batches, such that the latency is amortized over a number of queries. Currently, the demonstration application requests log probabilities for each token individually. This could be modified so that log probabilities for all tokens in a sentence can be requested at once; or alternatively, the algorithm could be further optimized to score multiple sentences at once, thus allowing even larger query batches. This solution has been implemented with great successful in distributed language models [31, 12, 2]. Since memcached provides an efficient mechanism for handling batch requests, this technique will dramatically improve throughput.

Throughput can be further increased by switching from TCP queries to UDP queries (which memcached supports), although this forgoes the service guarantees provided by TCP. However, let us not get lost in these optimization details: the bottom line is that the performance of the current implementation suffices for the applications that motivated this work in the first place, and for applications that may require even higher levels of performance, relatively straightforward optimizations can further increase query throughput significantly.

7.2 Architectural Considerations

There are a number of configurations that we have not had an opportunity to explore, which we leave for future work. In the current setup, each worker in the cluster serves double-duty for running mappers/reducers as well as serving memcached requests. Memcached is not processor intensive, but the memory used for caching takes away memory from the mappers and reducers themselves. Alternatively, one could imagine a cluster configuration with separate MapReduce workers and memcached servers. How does the performance of this setup compare? Given a fixed amount of hardware resources, is there an optimal allocation of different roles for cluster nodes? This may naturally depend on the nature of the algorithms, but it may be possible to develop generalizations based on generic load profiles.

In our present implementation, we have not paid particular attention to the issue of robustness, which is of course a major feature of the MapReduce framework. Currently, if a memcached server is unreachable, the requests for log probabilities to that particular server will time out and trigger an exception, which (after retries) eventually causes the entire job to fail. Thus, the current architecture provides job-level correctness guarantees, which may in fact be adequate for clusters of moderate size (where the likelihood of a machine failure during any particular Hadoop job is small). In the current implementation, error recovery simply requires rerunning failed jobs and avoiding unavailable memcached servers. Since most real-world applications consist of multiple MapReduce jobs, this is equivalent to a checkpointing mechanism. Of course, robustness could be achieved by replicating memcached copies and modifying the client API accordingly—this will also increase throughput by distributing load across the replicas. However, with a fixed amount of computing resources, it would be necessary to balance cache capacity with replication for both robustness and throughput. We leave exploration of this tradeoff for future work.

8 Conclusion

A large class of MapReduce algorithms in Hadoop require low-latency, high-throughput access to static global resources. Examples of problems that do not have satisfactory solutions at scale today include querying language model probabilities, accessing model parameters in iterative algorithms, or performing joins across relational datasets. In this paper, we have shown that memcached, an open-source distributed in-memory object caching system previously used in database-driven Web application, can be adapted for Hadoop and provides a feasible, scalable, and general-purpose solution to the problems discussed above. To our knowledge, this application is novel, and expands the class of problems that can be practically tackled in Hadoop. Use of an existing commercial-grade, open-source tool means that our solution can be rapidly deployed in both research and production environments.

9 Acknowledgments

This work was supported in part by the NSF under award IIS-0836560. Any opinions, findings, conclusions, or recommendations expressed in this paper are the authors' and do not necessarily reflect those of the sponsor. We would like to thank Yahoo! for leading the development of Hadoop and Amazon for EC2/S3 support. This paper benefited from helpful comments by Philip Resnik and discussions with Chris Dyer. The first author is grateful to Esther and Kiri for their kind support.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, Bolton Landing, New York, 2003.

[2] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 858–867, Prague, Czech Republic, 2007.

[3] P. F. Brown, V. J. Della Pietra, S. A. Della Pietra, and R. L. Mercer. The mathematics of statistical machine translation: parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI 2006)*, pages 205–218, Seattle, Washington, 2006.

[5] D. Chiang. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL 2005)*, pages 263–270, Ann Arbor, Michigan, 2005.

[6] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pages 281–288, Vancouver, British Columbia, Canada, 2006.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, California, 2004.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 205–220, Stevenson, Washington, 2007.

[9] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistics Society*, 39(1):1–38, 1977.

[10] C. Dyer, A. Cordova, A. Mont, and J. Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *Proceedings of the Third Workshop on Statistical Machine Translation at ACL 2008*, pages 199–207, Columbus, Ohio, 2008.

[11] T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with MapReduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL 2008), Companion Volume*, pages 265–268, Columbus, Ohio, 2008.

[12] A. Emami, K. Papineni, and J. Sorensen. Large-scale distributed language modeling. In *Proceedings of the 32nd International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2007)*, pages IV–37–IV–40, Honolulu, Hawaii, 2007.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 29–43, Bolton Landing, New York, 2003.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT 2008)*, pages 260–269, Toronto, Ontario, Canada, 2008.

[15] P. Koehn, F. J. Och, and D. Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT/NAACL 2003)*, pages 48–54, Edmonton, Alberta, Canada, 2003.

[16] M. Kruijf and K. Sankaralingam. MapReduce for the Cell B.E. architecture. Technical Report CS-TR-2007-1625, Department of Computer Sciences, University of Wisconsin, October 2007.

[17] J. Lafferty and C. Zhais. Document language models, query models, and risk minimization for information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2001)*, pages 111–119, New Orleans, Louisiana, 2001.

[18] J. Lin. Exploring large-data issues in the curriculum: A case study with MapReduce. In *Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics (TeachCL-08) at ACL 2008*, pages 54–61, Columbus, Ohio, 2008.

[19] J. Lin. Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with mapreduce. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP 2008)*, pages 419–428, Honolulu, Hawaii, 2008.

[20] A. Lopez. Statistical machine translation. *ACM Computing Surveys*, 40(3):1–49, 2008.

[21] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge, Massachusetts, 1999.

[22] F. J. Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics (ACL 2003)*, pages 160–167, Sapporo, Japan, 2003.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, Vancouver, Canada, 2008.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University, 1999.

[25] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1998)*, pages 275–281, Melbourne, Australia, 1998.

[26] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA 2007)*, pages 205–218, Phoenix, Arizona, 2007.

[27] A. Venugopal, A. Zollmann, and A. Waibel. Training and evaluating error minimization decision rules for statistical machine translation. In *Proceedings of the ACL 2005 Workshop on Building and Using Parallel Texts*, pages 208–215, Ann Arbor, Michigan, 2005.

[28] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. In *Proceedings of the 25th International Conference on Machine Learning*, pages 1184–1191, Helsinki, Finland, 2008.

[29] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, Beijing, China, 2007.

[30] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2001)*, pages 334–342, New Orleans, Louisiana, 2001.

[31] Y. Zhang, A. S. Hildebrand, and S. Vogel. Distributed language modeling for n-best list re-ranking. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing (EMNLP 2006)*, pages 216–223, Sydney, Australia, 2006.