

Full-Text Indexing for Optimizing Selection Operations in Large-Scale Data Analytics

Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil
Twitter
795 Folsom St.
San Francisco, California
@lintool @squarecog @kevinweil

ABSTRACT

MapReduce, especially the Hadoop open-source implementation, has recently emerged as a popular framework for large-scale data analytics. Given the explosion of unstructured data begotten by social media and other web-based applications, we take the position that any modern analytics platform must support operations on free-text fields as first-class citizens. Toward this end, this paper addresses one inefficient aspect of Hadoop-based processing: the need to perform a full scan of the entire dataset, even in cases where it is clearly not necessary to do so. We show that it is possible to leverage a full-text index to optimize selection operations on text fields within records. The idea is simple and intuitive: the full-text index informs the Hadoop execution engine which compressed data blocks contain query terms of interest, and only those data blocks are decompressed and scanned. Experiments with a proof of concept show moderate improvements in end-to-end query running times and substantial savings in terms of cumulative processing time at the worker nodes. We present an analytical model and discuss a number of interesting challenges: some operational, others research in nature.

Categories and Subject Descriptors

H.3.1 [Information Storage and Retrieval]: Content Analysis and Indexing

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

MapReduce, Hadoop, Pig, database optimization

1. INTRODUCTION

MapReduce [6] has recently emerged as a popular framework for large-scale data analytics. Among its advantages

are the ability to horizontally scale to petabytes of data on thousands of commodity servers, easy-to-understand programming semantics, and a high degree of fault tolerance. Many organizations have adopted Hadoop, the open-source implementation of MapReduce, which provides a low-cost, easy-to-deploy platform for large-scale data analytics, particularly on “dirty” datasets that may have missing/duplicate fields, invalid values, etc. A Hadoop-based analytics stack excels at processing such data due to its flexible support for user-defined functions and ability to cope with inconsistent, ill-formed, or non-existent schema (yet take advantage of schemas when available). Various tools built on top of Hadoop such as Pig [14] and Hive [18] provide higher-level languages for data analysis: a dataflow language called Pig Latin and a variant of SQL, respectively.

It has been argued that management of semi-structured and unstructured data represents the biggest opportunity and challenge facing the database community today [10, 19]. The growth of semi-structured and unstructured data far outpaces the growth of relational data. Due to phenomena such as social media and search, multi-terabyte collections of semi-structured data such as service logs are inexorably intertwined with free-form data such as search queries, item descriptions, emails, and other user-generated content. As a result, approaches to large-data analysis that seamlessly integrate structure and unstructured data processing are becoming increasingly critical. We take the position that any modern analytics platform must support operations on unstructured text as a first-class citizen, not as an afterthought. It naturally follows that optimizations on free-text operations are just as important as any other traditional optimization technique on purely relational data. This paper focuses on one such optimization for processing free-text fields.

It has been pointed out that Hadoop lacks many optimizations that are common in relational databases, and therefore suffers from poor performance on certain analytics tasks [15, 17]. In fairness, however, Dean and Ghemawat [7] provide a nice counterpoint, and there is growing interest in systems that adopt hybrid approaches [1, 13]. One clearly inefficient aspect of Hadoop-based processing is the need to perform a full scan of the entire dataset, even in cases where it is clearly not necessary to do so. In this paper, we show that it is possible to leverage a full-text index to optimize selection operations on text fields within records. The idea is quite simple: the full-text index informs the Hadoop execution engine which compressed data blocks contain query terms of interest, and only those blocks are decompressed and scanned. Experiments with a proof of concept show

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MapReduce'11, June 8, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0700-0/11/06 ...\$10.00.

moderate improvements in end-to-end query running times and substantial savings in terms of cumulative processing time at the worker nodes.

Contributions. This work motivates and illustrates the importance of text processing capabilities in a large-data analysis pipeline. Although our idea is not novel—the integration of full-text indexing in relational databases dates back over a decade [8, 2, 21]—to our knowledge, this represents the first attempt within the Hadoop framework. We discuss interactions with block-based record compression and present an analytical model that characterizes the performance of our approach—both refinements largely missing from previous work. We argue that this work represents more than a simple adaptation of a well-worn idea to “fashions of today”—there are a number of fundamental differences about the Hadoop stack that present interesting challenges: some operational, some research in nature.

Paper Organization. We begin with an overview of MapReduce/Hadoop and our usage scenario (complete with a brief sketch of the analytics stack at Twitter) in Section 2. Our approach for optimizing selection operations on free-text fields is discussed in Section 3, complete with experimental results and a simple analytical model. Related work and challenges are presented in Sections 4 and 5, where we argue that even though our idea is not novel, aspects of the Hadoop stack provide opportunities and challenges that are different from previous efforts to integrate full-text capabilities into relational databases. We conclude in Section 6.

2. BACKGROUND

2.1 MapReduce and Hadoop

MapReduce provides a simple functional abstraction that shields the developer from low-level system details such as inter-process communication, synchronization, and fault-tolerance. Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction for “mappers” (that specify the per-record computation) and “reducers” (that specify result aggregation). 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 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.

In this framework, a programmer needs only to provide implementations of the mapper and the reducer. On top of a distributed file system [12], the runtime transparently handles all other aspects of execution on clusters ranging from a few to a few tens of thousands of cores, on datasets ranging from gigabytes to petabytes. The runtime is responsible for scheduling, coordination, handling faults, and sorting/grouping of intermediate key-value pairs between the map and reduce phases.

Even though its functional programming roots date back several decades, Google is credited with developing MapReduce into a distributed, fault-tolerant processing framework. Hadoop is an open-source implementation of MapReduce in Java, complete with the Hadoop Distributed File System (HDFS),¹ which provides the storage substrate. It has gained widespread adoption as the core of an open-

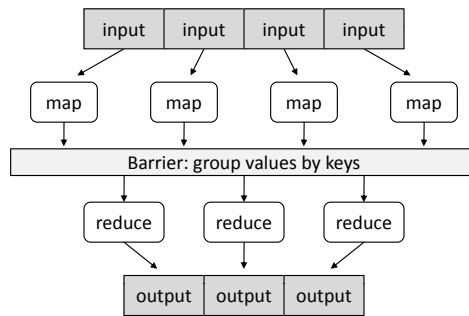


Figure 1: Illustration of the MapReduce framework: the “mapper” is applied to all input records, which generates results that are aggregated by the “reducer”. The runtime groups together values by keys.

source software stack for large-data processing and has become the focus of much recent work in the database community (e.g., [20, 14, 15, 1], just a name a few).

In HDFS, file blocks (typically 64 or 128 MB in size) are stored on the local disks of machines in the cluster (with a default replication factor of three). At job submission time, the Hadoop execution engine computes partitions of the input data, called input splits, which by default are aligned with the file block boundaries. Each input split is associated with a map task. To the extent possible, map tasks are scheduled on cluster nodes that hold a local copy of the block, thereby minimizing network traffic and taking advantage of the aggregate disk throughput of all nodes in the cluster—in this sense, code is moved to the data, not the other way around.

As each map task starts, the associated input split instantiates a record reader that reconstructs input key-value pairs from the underlying file stream. Critically, *all* records in the input dataset are read in every MapReduce job—even if the first thing that a mapper does is to apply a selection criterion to discard non-matching records. This work addresses this inefficiency in the case of selecting textual fields based on their content.

2.2 Twitter’s Analytics Stack

Twitter is a microblogging service through which users can send short, 140-character messages, called “tweets”, to their “followers” (other users who subscribe to those messages). Conversely, users can receive tweets from people they follow via a number of mechanisms, including web clients, mobile clients, and SMS. As of March 2011, Twitter has over 200 million users, who collectively post over 140 million tweets per day. We provide a brief overview of the Twitter analytics stack, which has previously been described in presentations at various technical forums.¹

A Hadoop cluster lies at the core of our analytics infrastructure. Data is written to HDFS via a number of real-time and batch processes, in a variety of formats. These data can be bulk exports from databases, application logs, network packet samples, machine health reports, and more. When the contents of a record are well-defined, records are serialized using one of two serialization frameworks. Proto-

¹<http://www.slideshare.net/kevinweil/hadoop-at-twitter-hadoop-summit-2010>

col Buffers [7] (protobufs for short) are a language-neutral data interchange format that supports compact encoding of structured data. They were introduced by Google and are used extensively within that organization. Recently, the framework has been partially released under an open-source license.² Thrift³ is a similar project from the Apache Foundation which provides a language-independent RPC mechanism in addition to serialization. The choice between Protocol Buffers and Thrift is largely up to data producers; the analytics stack we have created works equally well with both formats.

Encoded records are LZO-compressed. LZO compression offers two advantages: first, it provides a good tradeoff between compression ratio and speed; second, since the compression format consists of smaller individual compressed blocks (on the order of a couple hundred KB), a large LZO file can be split at block boundaries for processing by Hadoop. As part of the data ingestion pipeline, an automated process builds a block index for every LZO-compressed file. The index files are simple block number to byte offset mappings that are stored alongside the original data files. This information is used to align input splits to LZO block boundaries at the start of Hadoop jobs, which is automatically handled by the appropriate InputFormat.

Twitter stores many different types of records and logs in this serialized, LZO-compressed format. In a Hadoop job, different record types produce different types of key-value pairs for the map phase, each of which requires custom code for deserializing and parsing (as well as mechanisms to align input splits with LZO block boundaries). Since this code is both regular and repetitive, it is straightforward to use the serialization framework to specify the data schema, from which the serialization compiler generates code to read, write, and manipulate the data in a variety of languages. We felt it was desirable to enhance this code generation to further generate Hadoop-based record readers and writers for Protocol Buffer and Thrift schemas.

Twitter developed a project, called Elephant Bird,⁴ to automatically generate Hadoop record readers and writers for arbitrary Protocol Buffer and Thrift messages, LZO-compressed or otherwise. Elephant Bird also generates code for reading and writing compressed messages in Pig and Hive, as well as for working with other common data formats like JSON. The project, which like LZO and serialized messages underlies much of Twitter’s Hadoop-based stack, has been released under the Apache 2.0 license.

Instead of directly writing Hadoop code in Java, analytics at Twitter is performed almost exclusively using Pig, a high-level dataflow language that compiles into physical plans that are executed on Hadoop [14, 11]. Pig (via a language called Pig Latin) provides concise primitives for expressing common operations such as projection, selection, group, join, etc. This conciseness comes at low cost: Pig scripts approach the performance of programs directly written in Hadoop Java. Yet, the full expressiveness of Java is retained through a library of custom UDFs that expose core Twitter libraries (e.g., for extracting and manipulating parts of tweets). Pig also allows one to write custom loaders that read records into a specified schema. As mentioned

above, Elephant Bird has been extended with custom, code-generated loaders from Protocol Buffers or Thrift to Pig as well; an example is `StatusProtobufPigLoader` below.

Like many organizations, the analytics workload at Twitter can be broadly divided into two categories: large aggregation queries and exploratory *ad hoc* queries. The aggregation queries feed front-end report generation systems and online dashboards, and primarily involve scans over large amounts of data, typically triggered by our internal workflow manager. These are not the focus of this paper, as by nature they tend not to benefit significantly from indexing and other optimization techniques other than basic temporal partitioning—they need to scan and summarize all the available data.

Running alongside these large aggregation queries are *ad hoc* queries, e.g., one-off business request for data or machine learning experiments conducted by the research group. Although such jobs routinely involve processing large amounts of data, they are closer to “needle in a haystack” queries than aggregation queries. As a common case, Pig scripts begin with something like the following:

```
status = load '/tables/statuses/2011/03/01'
using StatusProtobufPigLoader()
as (id: long, user_id: long, text: chararray, ...);

filtered = filter status
by text matches '.*\\bhadoop\\b.*';
```

In this case, we are processing all statuses (tweets) on a particular day. The first Pig Latin statement loads the input data using a custom loader (which abstracts away from compression and protobuf encoding) and imposes a schema for convenient further access. The second statement is essentially a selection operation, where we only retain tweets that contain the term ‘hadoop’ (in the form of a regular expression that enforces word boundaries on both ends so that we don’t match embedded terms). Subsequent processing might count the number of tweets that contain links, identify other users mentioned in the set of matching tweets, aggregate by the clients from which the statuses originated (e.g., the Twitter website), group by timestamp to extract term occurrence time series, and so forth. As another example, while investigating phishing attacks we might wish to analyze traffic originating from a particular set of IPs, which could be accomplished by scanning HTTP logs and selecting those records matching an interesting set of IPs (by either a substring match or a regular expression match). Given the amount of web traffic received by Twitter, queries of this nature are processing intensive.

In most cases, given the nature of *ad hoc* queries, the selection criterion is highly selective, retaining only a small fraction of the entire dataset for subsequent processing. Yet, since Pig is simply a layer on top of Hadoop, all records must be scanned every time. This is inefficient and slow, particularly from a throughput perspective when there are multiple simultaneous queries of this form. It is this problem that we focus on here.

3. OPTIMIZING SELECTIONS

Our approach to optimizing selection operations involving free-text filters (e.g., regular expression matches) is quite simple and intuitive: we construct a full-text inverted index

²<http://code.google.com/p/protobuf/>

³<http://thrift.apache.org/>

⁴<http://github.com/kevinweil/elephant-bird>

on the field in question (at the LZO block level). Upon submission of a Hadoop job, the inverted index is consulted, and only those blocks that match the selection criterion are decompressed and scanned (more precisely, we process only those blocks known to contain at least one matching record of interest). Throughout this paper, we will illustrate with the running example of selecting tweets that match a particular keyword, with the understanding that the technique is broadly applicable to any type of record with free-text fields or other semi-structured data (for example, we could just as easily index HTTP logs by IP addresses).

3.1 Implementation

Based on the organization of our analytics stack, the appropriate granularity of full-text inverted indexes on free-text fields is at the LZO block level—critically, *not* at the level of individual records. Since LZO-compressed data must be read a block at a time, precisely pinpointing which record contains a particular keyword brings little additional benefit (beyond pinpointing the relevant block), since the entire LZO block must be decompressed anyway.

For each LZO block, we create a “pseudo document” consisting of the text of all tweets contained in the block. These pseudo documents are then indexed with Lucene, an open-source retrieval engine.⁵ To obtain compact indexes, we do not store term frequency information and term position information. Therefore, instead of simply concatenating all tweets together in each pseudo document, tweets are pre-processed to retain only one occurrence of each term and enumerated in lexicographic order. The structure of the index limits us to coarse-grained boolean queries. Lack of positional information precludes phrase queries, although experimental results suggest this isn’t a severe limitation. The tradeoff of query expressiveness for compact index structures appears to be a good choice.

Upon submission of a Hadoop job, the inverted index is consulted for blocks that meet the selection criterion; blocks are referenced by byte offset positions. The input splits of the dataset (i.e., partitions aligned with HDFS file blocks) are first computed as usual. Next, the start of each input split is advanced to the byte offset position of the first relevant LZO block within that split. The list of LZO blocks matching the selection criterion is then passed to the record readers assigned to each input split.

In a normal Hadoop job, record readers are instantiated on each of the worker nodes (via the TaskTrackers). Each record reader then sequentially scans the input split it was assigned to, decoding input records and passing them along to the mapper code. In our implementation, the record readers have been made aware of the LZO blocks matching the selection criterion (i.e., from consulting the inverted index). After processing a relevant LZO block, it skips ahead to the next matching block. Blocks known *not* to match the selection criterion are skipped, thus eliminating unnecessary disk IO and additional overhead necessary to decode records. Each relevant LZO block is processed as normal (i.e., the selection criterion is applied to each tweet).

In our current implementation, which is best characterized as a proof of concept, the developer specifies the selection criterion as part of configuring the Hadoop job. This design choice means that our optimization seamlessly inte-

	Query	Blocks	Records	Selectivity
1	hadoop	97	105	1.517×10^{-6}
2	replication	140	151	2.182×10^{-6}
3	buffer	500	559	8.076×10^{-6}
4	transactions	819	867	1.253×10^{-5}
5	parallel	999	1159	1.674×10^{-5}
6	ibm	1437	1569	2.267×10^{-5}
7	mysql	1511	1664	2.404×10^{-5}
8	oracle	1822	1911	2.761×10^{-5}
9	database	3759	3981	5.752×10^{-5}
10	microsoft	13089	17408	2.515×10^{-4}
11	data	20087	30145	4.355×10^{-4}

Table 1: Queries used in our evaluation, hand picked for different selectivity values. The third and fourth columns indicate how many LZO blocks and how many records contain each keyword, respectively.

grates with existing developer practices—if a selection criterion is specified, the input splits are adjusted and record readers are informed, as described above. Otherwise, the Hadoop job executes as normal. We note that as part of future work, selection optimizations can be automatically performed by analyzing plans from Pig scripts. Our Elephant Bird framework already generates custom Pig loaders automatically, and Pig provides a mechanism by which it can automatically push selection criteria from a script into a loader. The loader would then simply have to inject the criterion into the Hadoop job configuration as is done now explicitly. This is fairly straightforward and not critical to the performance of our optimization, so we did not implement the feature for the prototype discussed here.

3.2 Experimental Results

We present experimental results on the tweet stream for an arbitrarily selected day, August 1, 2010. On that day, 69.2 million tweets were recorded, totaling 6.07 GB compressed (beyond the actual tweets, each record stores related metadata). The compressed file contains 39767 LZO blocks, each 153 KB in size and containing 1740 records on average. The Lucene full-text index occupies 531 MB, or a bit less than a tenth of the size of the compressed dataset.

We selected a small set of queries by hand to represent a range of selectivity values, shown in Table 1. The fourth column shows the number of records (tweets) containing the term, and the third column shows the number of LZO blocks that contain matching records. Note that although our test set of queries contains only single-term queries, more complex boolean queries are possible (although not phrase queries). The actual queries are not particularly important, only the selectivity values they represent are. The experimental task was very simple: selection of tweets that match the relevant query term and writing those tweets to HDFS without any additional processing. This simple task isolates the impact of our selection optimization.

Experiments were run on a Hadoop cluster consisting of 87 nodes (each with dual quad core Xeons and 9 TB disk storage), running Cloudera’s distribution of Hadoop. Since this is a production cluster that is always running jobs, it is quite tricky to obtain reliable performance measurements. This is complicated by the relatively short running times of

⁵<http://lucene.apache.org/>

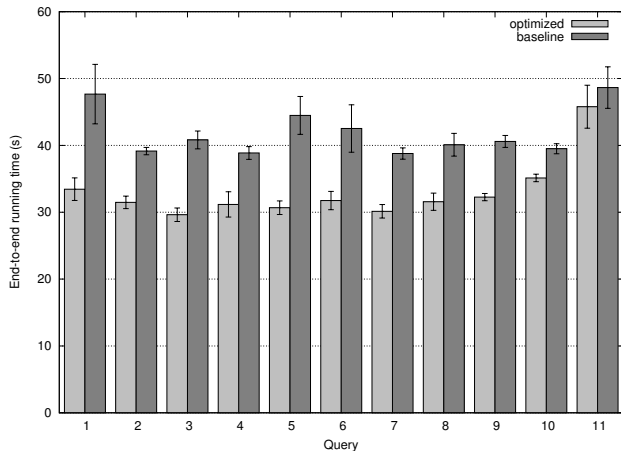


Figure 2: End-to-end job completion times for each of our queries, across 20 trials for each condition. Error bars show standard errors.

all jobs (< 1 minute). Our solution was to avoid peak hours and run many trials, hopefully smoothing out the “background noise” of concurrent jobs. Each experimental run consists of ten trials each of the baseline (brute force sequential scan) and optimized selection, alternating, for all 11 queries (220 trials total). We then repeated the experimental run (i.e., all 220 trials) again at a different time. We report aggregate statistics across *all* trials, including those that were clearly affected by concurrently running jobs (since due to our alternating trials, cluster load would affect each condition equally).

In all cases except for optimized selection with the queries ‘hadoop’ and ‘replication’, each job yielded 46 tasks (i.e., the input file spans 46 HDFS blocks). With those two queries, the jobs launched only 32 tasks—only those input splits contained a relevant LZO block.

Performance was measured in two ways: first, the end-to-end running time of the query; second, the cumulative time spent by all the mappers, which quantifies the total amount of work necessary to complete the job. This is accomplished by instrumenting the job with Hadoop counters. The end-to-end running times are shown in Figure 2 for each of the queries (in the same order as in Table 1). Error bars denote standard error across all of the runs. Running times for the optimized selection include consulting the Lucene inverted index to enumerate all matching LZO blocks, which takes less than a second on average. Figure 3 shows the cumulative running times, i.e., sum total across all map tasks. Error bars also show standard error.

We can see that optimized selection has a noticeable but not substantial effect on end-to-end running times. We explain this result as follows: First, Hadoop jobs are relatively slow in starting up, since the architecture was optimized for large batch processes. In fact, a significant portion of the processing time in both conditions is taken up by the fixed job overhead. Second, since our jobs are short-lived, it means that idiosyncratic interactions with concurrent jobs (i.e., in scheduling) can have a substantial impact on end-to-end running times. Finally, since the job occupies only a small fraction of the entire cluster capacity (46 tasks out of a total of 870 task slots across the cluster), all tasks run

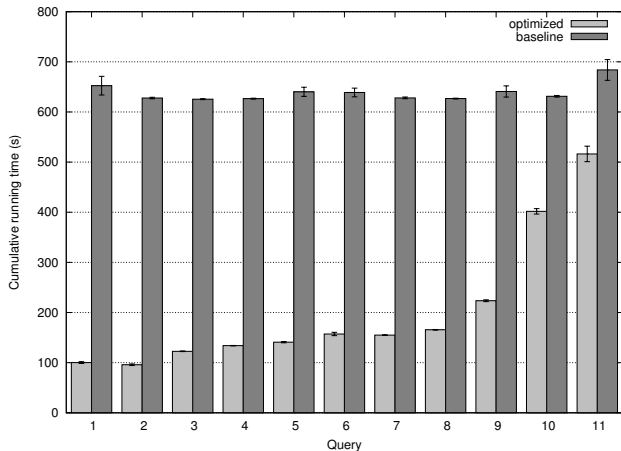


Figure 3: Cumulative running times for each of our queries (i.e., sum of amount of time spent in all the mappers), across 20 trials for each condition. Error bars show standard errors.

in parallel. From what we could tell, cluster capacity was not saturated during our experimental runs by concurrently running jobs. As a result, the end-to-end running time is bound by the *slowest* task. For substantially larger jobs, end-to-end performance will be bound by the throughput of task completion, and not simply task latency. Note that since we ran experiments on a production cluster, it was not practical to experiment with large jobs that might interfere with production processes. Nevertheless, for *ad hoc* analytics tasks at Twitter, it is common to process substantially larger datasets.

Despite these caveats, we still see noticeable reductions in end-to-end running times, ranging from approximately 30% for the most selective query and approximately 6% for the least selective query. However, it appears that for the least selective query in our testset, our selection optimization provides little benefit in terms of end-to-end running time (more on this later when we present our analytical model).

Turning our attention to the cumulative running time, i.e., the sum of running times across all mappers (Figure 3), we see significant reductions, varying with the selectivity of the query. This makes intuitive sense, since the amount of work saved by our optimization is directly related to the selectivity of the query. From Figure 3, we also get a sense of the job overhead of Hadoop—in the brute force case, each task runs for around 15 seconds, and in some of the optimized selection cases, each task runs for no more than a few seconds. The heavy overhead in job startup is a known issue in Hadoop, and is said to be somewhat mitigated in more recent distributions of the software.

Nevertheless, the conclusions from these experiments are fairly clear. Our selection optimization decreases end-to-end running time moderately for small, selective queries. However, the cumulative running time results suggest greater gains for larger queries and substantial increases for query throughput (i.e., with many concurrent queries).

3.3 Analytical Model

To complement our experimental results, we present a simple analytical model that predicts the number of com-

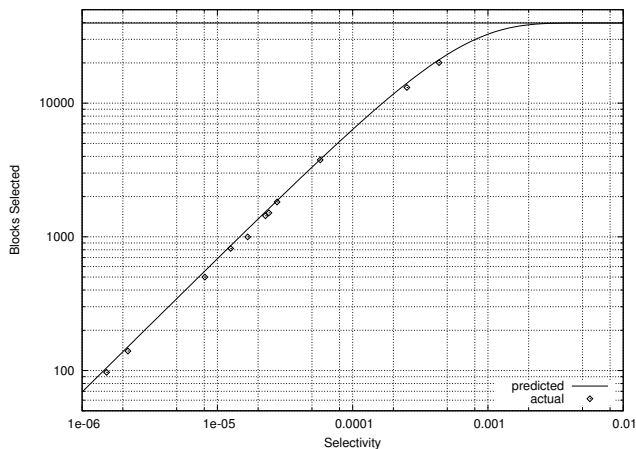


Figure 4: Poisson model of the number of compressed LZ0 blocks that must be scanned as a function of selectivity for our experimental conditions. The solid horizontal line represents the total number of blocks in the dataset. Empirically observed values are plotted as diamonds.

pressed LZ0 blocks that need to be scanned as a function of selectivity. We can model the occurrences of tweets with a particular term as a Poisson process. As such, the probability that we observe exactly k occurrences within a given window (i.e., k tweets containing the query term in an LZ0 block) is the following:

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!} \quad (1)$$

where λ is the expected number of occurrences within the given window, specified by the average number of tweets per block and the selectivity.

The fraction of blocks that need to be scanned is therefore $1 - f(k=0; \lambda)$. This is plotted in Figure 4 for our dataset, overlaid with empirically observed values from Table 1. The solid horizontal line represents the total number of blocks. We see that the Poisson model fits the actual data quite well, from which we can make predictions about the performance of our optimization across a variety of scenarios. At selectivity of 0.001, Hadoop must scan 82% of all blocks, and at 0.002, 97%. At those levels, our optimization becomes ineffective. Note that selectivity can be predicted *a priori* from the inverted index—and therefore the system can apply the optimization only when it is advantageous.

4. RELATED WORK

The integration of full-text indexing within relational databases is of course not a new idea—in fact, commercial offerings from Oracle, IBM, Microsoft, and others all provide this capability. Much work has been done along these lines, and here we discuss a few representative pieces. In one implementation [8], external search engines can be integrated into standard SQL queries by rewriting queries that contain search predicates into subselects, where the nested query calls out to the external search engine (much like a UDF). Agrawal et al. [2] provide a different approach by storing inverted indexes themselves as symbol tables, and rewriting queries in terms of these tables based on enumerating

join trees (see Yu et al. [21] for a recent survey of similar approaches, some of which require no knowledge of the database schemas). Our work is different from two perspectives. First, from a technical perspective, integration of full-text indexes into the Hadoop environment presents several challenges that are not present in relational databases (we specifically address this point in Section 5). Second, from an applications perspective, we believe that previous systems have not persuasively argued the “business case” for integrating full-text indexing capabilities with relational queries, which may partially explain the lack of commercial success in the marketplace. In what scenario would users need to issue free-text queries against an RDBMS? Certainly not OLTP, and these existing techniques also appear ill-suited for OLAP. In contrast, we provide a compelling real-world scenario—business needs that Twitter faces daily.

There is, of course, plenty of research that has attempted to bridge relational databases and MapReduce (and related) programming models. Examples include an extension of the original MapReduce model called MapReduceMerge [20] to better support relational operations, HadoopDB [1], an architectural hybrid that integrates Hadoop with PostgreSQL, and Dremel [13], which takes advantage of columnar compression for large-scale data analysis.

Our work is closest to MANIMAL [3], which also optimizes selection operations in Hadoop programs. However, the system adopts a different approach based on static code analysis. MANIMAL takes as input a compiled Hadoop program with input and associated parameters, runs through distinct analysis and optimization phases before submitting the optimized plan to an execution fabric. While this is a more general framework, it is more heavyweight and less tightly integrated into Hadoop as it currently exists. In contrast, our approach does not modify Hadoop itself—all code changes are limited to existing APIs provided by the framework—which makes it closer to being production ready. Another interesting point of comparison is Hadoop++ [9], which injects *trojan indexes* into Hadoop input splits at data loading time. These indexes make it possible to execute relational operations efficiently, without altering Hadoop internals (as in our approach). However, these trojan indexes are not designed for full-text processing, which is the focus of our work.

5. CHALLENGES AND FUTURE WORK

Fundamentally, the design of the Hadoop stack is at odds with the requirements of full-text search. HDFS explicitly trades low-latency random access for high-throughput streaming reads, since the framework was specifically designed for batch processing. Yet low-latency random access is exactly what characterizes the IO patterns of full-text search—which requires fast access to (relatively short) postings lists to compute a results list. This remains an unresolved problem in our current implementation, where the Lucene inverted indexes are stored on the local disk of the gateway node from which Hadoop jobs are submitted. Since input splits are computed from the job submission node, this is a workable short-term solution, but fails to address long term needs (e.g., fault tolerance and scalability).

Google uses Bigtable [4], a sparse, distributed, persistent multi-dimensional sorted map, to hide latencies associated with GFS, their distributed filesystem. Recent improvements to Bigtable include coprocessors [16], a feature

similar to “triggers” found in traditional databases, which provides better support for incremental processing. It is further known that inverted indexes for the Google search engine are completely served from memory [5]. This combination appears to be a workable solution, but the best approach for integrating full-text search capabilities in the Hadoop ecosystem remains an open question. The open-source implementation HBase⁶ lags in performance, stability, and feature set compared to Bigtable. Even implementation and maturity issues aside, a distributed storage system (that underlies both Bigtable and HBase) seems ill-suited for storing postings lists that need to be accessed rapidly. Serving indexes from memory alleviates this mismatch, but imposes a resource requirement that is impractical for many organizations with more modest resources.

One major difference between our work and similar work in the context of RDBMSs is that databases provide mechanisms for tighter coupling of components (e.g., index updates, query rewrites, etc.). In the Hadoop context, looser coupling of components presents a number of challenges, primarily operational in nature. Whereas in a database, central metadata is maintained for all components, the connection between them in Hadoop is tenuous, enforced by fragile mechanisms like file ownership, access permission rules, and (perhaps unwritten) convention. A lot of effort must be spent when building these types of indexing solutions just to map indexes to the data that was indexed, as files may be renamed, deleted, moved, or replaced with newer versions without appropriate coordination.

Even more fundamentally, Hadoop requires an external framework for managing data and process dependencies: e.g., after successful ingestion of data, some process must trigger the building of block indexes and error checking mechanisms. Although Twitter has internally developed such a system, and other open-source solutions such as Oozie⁷ exist, it is not entirely clear that managing complex data dependencies robustly in production environments is a solved problem for Hadoop in general. There is certainly no consensus on best tools or even best practices among practitioners. In this respect, the Hadoop community needs to accumulate more experience.

In the current prototype, construction of indexes is a process that must be explicitly set up by an administrator, who provides the Protocol Buffer or Thrift definition, encodes field extraction and tokenization rules, and provides the requisite InputFormat classes to be used when performing optimized scans as described in this paper. A more robust system would allow for automated indexing of ingested data, provide a scalable solution for storing and querying such indexes, and integrate with query layers like Pig and Hive natively in a way that would allow the end user to take advantage of existing indexes transparently, perhaps even creating them on an as-needed basis. These issues, as well as the more general question of how best to apply full-text indexing capabilities to improve common analytics tasks, will be addressed in our future work.

6. CONCLUSIONS

Recognizing the growing importance of unstructured free text within data management systems, we present and eval-

uate a simple optimization for selections on free-text fields for analytics applications. Heeding the call that the management of unstructured data represents the biggest opportunity and challenge facing the database community today, this work takes a small step forward.

7. ACKNOWLEDGMENTS

We’d like to thank the Twitter analytics team for their contributions to the analytics stack, and specifically the following who contributed to Elephant Bird: Raghu Angadi, Ning Liang, Chuang Liu, Florian Liebert, and Johan Oskarsson. The first author is grateful to Esther and Kiri for their loving support and dedicates this work to Joshua and Jacob.

8. REFERENCES

- [1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *VLDB*, 2009.
- [2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. *ICDE*, 2002.
- [3] M. Cafarella and C. Ré. Manimal: Relational optimization for data-intensive programs. *WebDB*, 2010.
- [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. *OSDI*, 2006.
- [5] J. Dean. Challenges in building large-scale information retrieval systems. *WSDM Keynote*, 2009.
- [6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.
- [7] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *CACM*, 53(1):72–77, 2010.
- [8] S. Dessoach and N. Mattos. Integrating SQL databases with content-specific search engines. *VLDB*, 1997.
- [9] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *VLDB*, 2010.
- [10] A. Doan, J. Naughton, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong. The case for a structured approach to managing unstructured data. *CIDR*, 2009.
- [11] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of MapReduce: The Pig experience. *VLDB*, 2009.
- [12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SOSP*, 2003.
- [13] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *VLDB*, 2010.
- [14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. *SIGMOD*, 2008.

⁶<http://hbase.apache.org/>

⁷<http://yahoo.github.com/oozie/>

- [15] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. *SIGMOD*, 2009.
- [16] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. *OSDI*, 2010.
- [17] M. Stonebraker, D. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: Friends or foes? *CACM*, 53(1):64–71, 2010.
- [18] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive—a petabyte scale data warehouse using Hadoop. *ICDE*, 2010.
- [19] G. Weikum. DB & IR: Both sides now. *SIGMOD Keynote*, 2007.
- [20] H. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. *SIGMOD*, 2007.
- [21] J. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *Data Engineering*, 33(1):67–78, 2010.