# Manimal: Relational Optimization for Data-Intensive Programs

Michael J. Cafarella
University of Michigan
Ann Arbor, MI 48109-2121
michjc@umich.edu

Christopher Ré
University of Wisconsin
Madison, WI 53706-1685
chrisre@cs.wisc.edu

## ABSTRACT

The MapReduce distributed programming framework is very popular, but currently lacks the optimization techniques that have been standard with relational database systems for many years. This paper proposes MANIMAL, which uses static code analysis to detect MapReduce program semantics and thereby enable wholly-automatic optimization of MapReduce programs. For example, a programmer's *map* function that emits data only when an `if...` statement holds true is essentially encoding a selection condition; code analysis can detect and characterize these conditions. If MANIMAL has an appropriate index available, it can then alter MapReduce execution to use it.

MANIMAL can address many different optimization opportunities, including projections, structure-aware data compression, and others. However, this paper illustrates the system by focusing on one: efficient selection. We give a static analysis algorithm that can detect selections in user programs, and cover how MANIMAL can employ a B+Tree to execute these selections efficiently at runtime. Testing MANIMAL on several standard MapReduce programs, we show that selection alone can automatically reduce a standard program's runtime to 63% of conventional MapReduce execution time on identical hardware. We also give an in-depth discussion of other optimization targets and detection techniques.

## 1. INTRODUCTION

MapReduce is a popular framework for data-intensive programs that allows programmers to write large distributed jobs in a familiar UNIX-style development environment. Developers can use common programming languages such as Java and C++, the programs operate over bytestream-oriented files rather than relations, and developers do not need to formally declare a schema. The MapReduce framework itself handles large-scale distribution and error-handling tasks. Moreover, at least one MapReduce implementation (Hadoop) has been found to be very easy to setup and configure when compared to current distributed relational databases [11].

However, MapReduce programming comes at a cost: Pavlo *et al.* [11] showed that a MapReduce program can run 2-50x slower than a relational query that executes the same task on the same hardware. MapReduce programs can make up for this inefficiency by using massive numbers of machines, but they remain very inefficient, posing heavy financial and other costs. From the literature, one may conclude that a developer is forced to make a choice: to opt for the strengths of the MapReduce programming model or to opt for the efficiency of an RDBMS – but not both. In this work, we argue that there is a middle ground: we can combine the MapReduce programming model with many of the efficient execution techniques commonly used by RDBMSes. Our MANIMAL[1] system allows MapReduce programs to run unchanged, while reducing runtime to 63% of previous execution time using a single optimization type.

A key challenge in building MANIMAL is in safely applying optimizations to standard unmodified MapReduce programs. The central novelty of our work is the application of static analysis-based techniques to enable relational-style optimizations in a MapReduce setting. The individual components are straightforward: the analysis involves primarily known tasks such as code-reachability testing, and the optimizations are drawn from past database research. As we describe in Section 2.1 below, this approach poses some limitations on the kind of programs that MANIMAL can successfully optimize. However, one hypothesis of this work is that while programmers *can* write near-arbitrary code, in practice they tend to employ a small number of familiar programming idioms. This is partly borne out by a brief survey of MapReduce programs (also discussed below). Moreover, we have implemented MANIMAL and we are applying it to real programs today. We have ideas for extending it to additional program types in the future.

From a developer's perspective, MANIMAL is almost identical to conventional MapReduce implementations such as Hadoop. However, the internals of MANIMAL are quite different. First, while a conventional MapReduce system operates strictly on the input files indicated by the user, MANIMAL may choose alternate forms of the data (such as indexes). Second, standard MapReduce always scans every byte of the input data, while MANIMAL attempts to reduce work by only processing bytes that actually matter for the user's output. Finally, MapReduce systems today execute exactly the compiled bytes of the user's program, but MANIMAL may choose

[1] MANIMAL takes a hybrid approach to RDBMS and MapReduce systems, which is reflected in its name.


*WebDB* '10 Indianapolis, IN USA

to instead execute code that produces identical results but is more efficient to run.

*Other MapReduce Research.* There has been a large amount of recent interest in MapReduce systems. Several research efforts have explored the problem of scheduling task execution [8, 14]. Other researchers examined how to manage a set of machines that run multiple different MapReduce-style cluster tools simultaneously [7]. Afrati and Ullman [3] investigated how to efficiently perform joins using the MapReduce framework. Yang, *et al.* [13] extended the programming model to *Map-Reduce-Merge*, allowing the user to express different join types and algorithms.

None of the work cited here directly addresses the optimizations that we propose with MANIMAL. Moreover, we are not aware of any work that proposes a framework for detecting and applying a large number of optimizations to MapReduce execution. Indeed, some of the above systems could be integrated with MANIMAL, given some modifications.

One system that initially appears to be quite related to MANIMAL is HadoopDB [2], which attempts to combine relational and MapReduce qualities into a single system. However, HadoopDB has very different goals from MANIMAL. HadoopDB is an effort to build a highly-scalable parallel database, using a large number of single-machine relational databases tied together by a MapReduce-based communications layer. It is a parallel relational database in which MapReduce is used to address some scaling issues. HadoopDB users write SQL queries, not arbitrary MapReduce programs, so HadoopDB's focus did not fall on the problems associated with ordinary MapReduce program execution.

There are some systems, such as Pig [10], that compile higher-level languages into MapReduce for execution. One may argue that instead of optimizing existing MapReduce programs, a better approach would be to simply target a higher-level language. Although we agree that Pig and similar systems have wide application and that optimizing their outputs would be useful, we think that MapReduce programs themselves are very appealing to users for developer-interface reasons. Moreover, we believe our lower-level approach will be useful in the future for a range of MapReduce optimization tasks, including optimizations that address multiple jobs simultaneously; it would be surprising to us to find that these higher-level languages ever constitute *all* of the jobs on a highly-used cluster.

*Organization and Contribution.* The central contribution of this work is a framework for optimizing MapReduce programs that allows us to apply many techniques learned from decades of relational database research. By employing static analysis to automatically discern some of the semantics of user code, MANIMAL can operate on unchanged compiled MapReduce programs. For clarity of exposition, we present full details only for the selection operator. Other operators follow a similar structure, with appropriate modifications. We show that this one optimization can automatically yield a program that runs in 63% of the time required by a standard MapReduce system. We also briefly describe how MANIMAL would work with several additional operators.

The remainder of this paper is organized as follows. In Section 2, we describe a typical execution of the MANIMAL pipeline when applied to a small and well-known MapReduce program. Section 3 shows MANIMAL's performance impact. Finally, we discuss future optimizations and conclude in Sections 4 and 5.

# 2. MANIMAL BY EXAMPLE

In this section we describe precisely how the components from Figure 1 work together to handle a specific program. In addition to the **analyzer**, **optimizer**, and **execution fabric** components, MANIMAL also uses the *user's program*, the **catalog**, and various *on-disk files*. We will process the following *map* function, which is similar to a MapReduce program described by both Dean and Ghemawat [6] in their original MapReduce paper, and by Pavlo, *et al.* [11], in their recent comparison of MapReduce and RDBMS performance.

```
public void map(Text key, Text value,
         OutputCollector<Text, LongWritable> output)
         throws IOException {
  Matcher matcher = pattern.matcher(value.toString());
  while (matcher.find()) {
    output.collect(new Text(matcher.group(group)),
                   new LongWritable(1)); } }
```

This function encodes a simple grep count program, counting the number of lines in a text file that match the regular expression given by `pattern`. Like `pattern`, `group` is a runtime user parameter that controls the region of the regular expression match that is emitted. The reduce function simply sums the emitted `1` values. We can run this on any text file. Note that the program has been slightly edited here for clarity.

If, for a given *map* input pair, the while statement in the *map* code never evaluates to true, then it will not pass any data to the shuffle/sort/reduce phases. That is, it expresses a selection test on the **value** passed into the *map* function, which determines if anything is actually emitted. We will now show how MANIMAL can automatically detect, and take advantage, of this selection test in order to improve the program's execution time.

## 2.1 Step 1: Analysis

Most of MANIMAL's novelty lies in the **analyzer**, whose job is to determine the set of optimizations that are safe to apply to a MapReduce program. It takes as input the 4-tuple $(C, M, I, P)$, where:

- $C$ is the complete compiled program, including all methods that the program may invoke. In this case, it is simply the above program.
- $M$ is the "main" method of the program, where execution begins. *E.g.*, the `main()` function of `org.apache.hadoop.examples.Grep`.
- $I$ is the input file path for the program. In this case, it indicates the text file to be processed.
- $P$ is a set of parameters, in the form of a set of $(attr, val)$ pairs. In this case, one such pair might describe the regular expression used to initialize `pattern`; *e.g.*, `mapred.mapper.regex` might contain a regular expression that matches all strings that start with `b`.

Note that for the Hadoop implementation of MapReduce, the above information can be collected automatically as part of the job submission process.

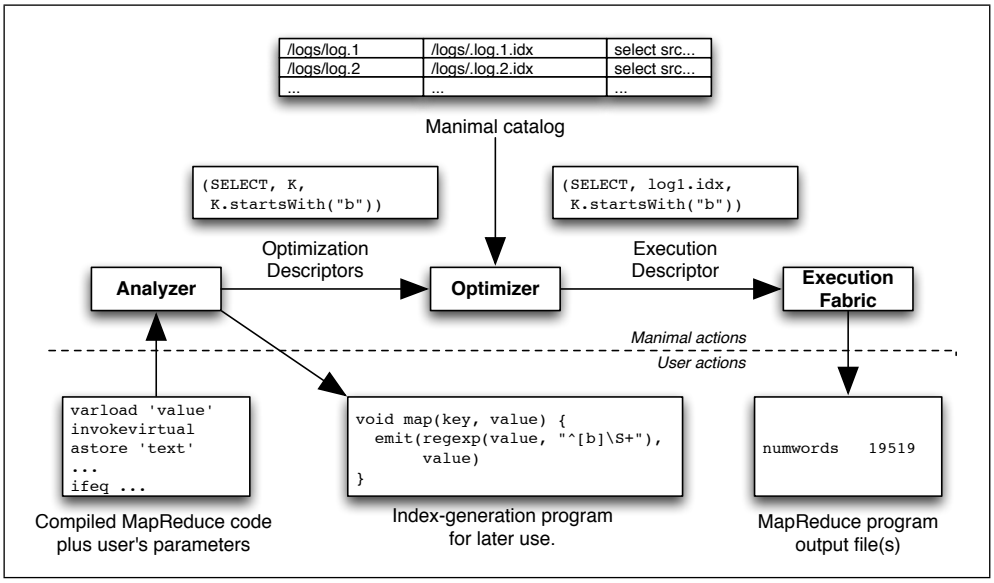


**Figure 1: The three stages of MANIMAL program execution. The user submits a compiled MapReduce program to the analyzer, which sends output to the optimizer, which emits instructions for the execution fabric.**

The main task of the **analyzer** is to produce a set of *optimization descriptors*. These descriptors enable MANIMAL to carry out a phase roughly akin to logical rewriting of query plans in a relational database. The basic "plan" that is manipulated is the generic MapReduce execution pipeline of map-sort-shuffle-reduce. The descriptors characterize a set of potential modifications that remain logically identical to the original plan.

In the case of selection optimization, the descriptor contains a logical expression that indicates when the *map* function must be executed in order to produce the same output as the user's original program. This expression uses only constants and a key value pair $K', V'$. This expression evaluates to true in all cases where the user's original map function emits a `key, value` pair. For example, the grep program's *map* function from the start of this Section yields a boolean condition of `(K' startsWith 'b')`.

The **analyzer** also emits an *index generation program* that can be applied to the input file $I$. It transforms $I$ into a new output file $I'$, in which each input pair $K, V$ is turned into $K', V'$. In the case of selection optimization of the above program, the index generation program yields a B+Tree that maps the first word of each line to the original full line of text. This allows the system to efficiently find all entries where the boolean expression `(K' startsWith 'b')` holds. Running the program also adds an entry to the **catalog**.

The **analyzer**'s full output consists of the input tuple, plus the set of optimization descriptors $O$ and index generation program $X$.

We now focus on how the **analyzer** actually obtains semantic information about the user's program, which is critical to the entire MANIMAL project.

*Static Analysis.* MANIMAL's **analyzer** attempts to be as *complete* as possible when searching for optimizations, while making only optimization decisions that are definitely *safe*. In other words, it is acceptable - although regrettable - when the **analyzer** misses a good optimization opportunity, but it is unacceptable to find an incorrect optimization that leads to logically-wrong output.

The **analyzer** is a best-effort system that will not always find possible optimizations. Note that traditional optimizing compilers, which have been very successful, are similar. It will always be possible for programmers to write code in a way that sidesteps optimization, but in practice we believe few will do so. We plan to expand the **analyzer** to handle new programming idioms and optimization opportunities as we observe them in real-life programs.

Detecting the selection optimization in the grep program's *map* function involves two major phases. The first is to compute a logical expression that must hold whenever the function emits a tuple.

1. Obtain the control-flow-graph of *basic blocks* in the function. A basic block is a region of code in which there are no flow-of-control transitions; all such transitions happen between blocks. Finding the control-flow-graph is a well-known technique drawn from the programming language community [4].

2. Enumerate all basic blocks that emit a `key, value` pair for the next stage of MapReduce processing. This simply involves examining the instructions in each block.

3. For each such block, trace a path in the graph of basic-blocks back to the entry point of the function. For each path, accumulate all of the conditional tests that permit access to each block in the path, thus yielding a conjunction of logical expressions. By examining all the blocks that emit `key, value` pairs, we obtain a list of these logical conjunctions. If *any one* of the expressions in this list is true - *i.e.*, when a disjunction of them is true - the *map* function must be executed.

The resulting disjunction describes when the user's code must be executed. If MANIMAL can detect *map* inputs under which the disjunction must be false, then we can safely skip *map* execution for those values. For example, in the code we are analyzing, we can find that invoking *map* only has an impact when `matcher.find()` is true.

The second phase is to examine these logical expressions in more detail. In particular, we want to describe the components of the expressions in terms of a handful of atoms: program constants, user-given parameters, and `key, value` inputs to *map*.

1. Compute a directed acyclic graph representation of the code in the *map* function. This, too, is a standard programming languages technique [4]. Each node in this graph is a statement in the program. An edge between nodes describes when a node has as an operand the output of a previous statement.

2. For each portion of the logical test found in the first phase, find the corresponding statement in the DAG. Follow the chain of references as far "up" in the program as possible. If this chain contains only *functional* method calls, it will eventually terminate in the entry node for the method. The edges connected to the entry node will include some combination of program constants, user parameters, and the *map*'s `key, value` inputs. A functional method call is one in which the output depends strictly on its inputs. This upward analysis may extend to code in the initializer for the user's program (not seen in the example code here).

3. Test whether the condition is one that can be efficiently handled by one of the available optimization techniques. Each technique has its own optimization-specific test.

The second phase plays out as follows on our example:

1. Consider the DAG representation of the `while` test. The node that corresponds to this test points to the invocation of `matcher.find()`. That statement, in turn, points to the instantiation of the `matcher` on the previous line, which points to the invocation of `pattern.matcher()`.

2. We find that the `matcher.find()` test involves `pattern` and `v`. The value `value` is given to the *map* function. The `pattern` object is initialized (in the unseen constructor) using parameter `mapred.maper.regex`. The **analyzer** has built-in information about Java regular expression methods, and knows that these calls are all functional.

3. In general, regular expression testing cannot be optimized using a B+Tree index. (Though as mentioned, other indexing techniques may apply.) However, in the current case we have a regular expression that tests the value of the first word in the string, so MANIMAL can employ a B+Tree.

When the above **analyzer** sequence succeeds - the optimization is safe and there is an available optimization technique - MANIMAL emits the *descriptor* and *index generation program* for the next step in the pipeline. This **analyzer** output is critical in obtaining the performance gain on this example program that we describe in Section 3.

*Analysis Limitations.* The two **analyzer** flaws that concern us are cases where the output may be unsafe, and cases where the **analyzer** fails to pick up a possible optimization.

In order for the selection-detection system here to be unsafe, it must decide to avoid invoking *map* when doing so is in fact called for. One possibility arises when the programming language can `goto` an arbitrary location computed using address arithmetic. In such a case, the original program may execute code for a block even though the logical conditions associated with it in the *map* do not hold true.

Another unsafe condition might arise if MANIMAL were to incorrectly evaluate the conditional guard for a basic block. This might happen if a basic block's entrance condition decision employs non-functional method applications that MANIMAL believes, in fact, to be functional. Luckily, this is not a real concern - MANIMAL is extremely conservative with tests of whether a method is *functional* or not. Right now MANIMAL uses a hard-coded list of functional methods, such as certain `String` methods and several relating to regular expressions. Methods not on that list are assumed to be non-functional. This approach is flawed, as we cannot expect to inspect many code libraries by hand. In the future, the **analyzer** will attempt to detect functional methods directly by examining source code, removing the need for a handmade list.

A different problem is that the **analyzer** will simply fail to pick up on potential optimizations. Because the MANIMAL **analyzer** is conservative, it may fail to recognize optimization opportunities that are in fact safe. For example, the **analyzer** does not currently permit a selection optimization if the conditional test involves a public static variable. This policy exists because external code can silently modify the static variable, and thus change whether *map* must be executed. The eventual solution is to apply the **analyzer** to every byte of coded in the program and explicitly test for such modifications (and not limit the **analyzer** to a small set of *map*, *reduce*, and related methods).

A brief survey of some MapReduce programs shows that our current simple **analyzer** can already find many selection optimizations. There is not a large amount of open-source MapReduce code to examine, but we found two sources: material released by Pavlo, *et al.* [11]; and the Mahout project [9] to build machine-learning algorithms on top of MapReduce. Of the four benchmark programs from Pavlo, *et al.*, two include a selection-style test in the *map* function. Of a random sample of 12 sub-projects drawn from Mahout, we found two that contain a selection test. Of these programs, **analyzer** correctly detected three of the four optimization opportunities. (It did not detect any unsafe ones.)

The one that MANIMAL missed is interesting. The "Benchmark 4" program from Pavlo, *et al.*, uses a regular expression to parse its `value` input into a series of urls, counting each unique url by keeping a url-count map in a hashtable. That done, it then iterates through the hashtable, emitting each `url, count` pair for the *reduce* function. This program could be optimized by only executing the function for inputs that the regular expression can parse correctly. Our current **analyzer** misses any selection opportunity. Because it does not have any built-in knowledge of the Java `Hashtable` class, and because it does not attempt to examine the `Hashtable` bytecode directly, the **analyzer** cannot know that a call to `Hashtable.keySet()` returns results only when they have been previously inserted.

We can address this issue in the future either by giving MANIMAL built-in knowledge of `Hashtable`, or by directly examining the bytecode for `Hashtable` and computing

the influence of `key, value` outside the boundaries of *map*. However, note that it is also an example of a larger class of problems that eventually go beyond what we can reasonably expect from static analysis. If the *map* function were to store the `url, count` pairs on a remote network service instead of a `Hashtable` object, there is nothing the **analyzer** could do.

The output of the **analyzer** contains the original 4 inputs, plus the index generation program $G$ and the optimization descriptors $O$.

## 2.2 Step 2: Optimization

The **optimizer** uses the **analyzer**'s output, plus information in the **catalog**, to determine an execution plan for MANIMAL. Its rough relational analogue would be a very simple rule-based optimizer. The **catalog** is a simple mapping from a filename to zero or more $(X, O)$ pairs, where $X$ is an index file, and $O$ is an optimization descriptor. It is built out of records added by executions of index generation programs. For example, the **analyzer** for a user program might yield an index generation program that, when run on *words.txt*, emits the index file *words.txt.idx*, and then adds

```
(words.txt, words.txt.idx, (selection, K' startsWith b))
```

to the **catalog**. The **optimizer**'s output contains all of its input, plus a recommended execution plan $X$.

The **optimizer** examines the **catalog** to see if there is any entry for input file $I$. If not, then $X$ simply indicates that MANIMAL should run the unchanged user program on $I$, just as a standard MapReduce system would. If there is at least one entry for the input file, and a **catalog**-associated optimization descriptor is compatible with **analyzer**-output $O$, then the **optimizer** can choose an execution plan that takes advantage of the associated index file. The test for index-compatibility is dependent on the optimization being considered. For example, in the selection case, a B+Tree index that is associated with `(K' startsWith [bc])` is compatible with the **analyzer**'s boolean condition `(K' startsWith b)`. An index computed using `(K' startsWith c)` would not be.

The output plan is also optimization-specific. Our current implementation of optimization for selection uses a B+Tree. The plan indicates an index file to load, and that the program should process any stored data that matches the condition. (A differnt implementation might use an index for general regular expressions, such as that described by Cho, *et al.* [5].)

The user makes the decision whether to run the index generation program at all. Of course, index creation entails iterating over the entire input dataset and so many optimizations can be quite costly. The user may require a workload that requires many scans over the input data for index-computation to be worth the cost. However, this decision is not unique to MANIMAL; rather, it is a fact of life with any index-based optimization approach.

Currently, the **optimizer** is very basic. As the number of optimizations in MANIMAL increases, and the difficulty in choosing a high-quality plan $X$ increases, we plan to move to a cost-based model.

## 2.3 Step 3: Execution

The **execution fabric** receives a tuple that contains the execution plan $X$ and all of the output from the **analyzer**.

| Regular Expression | Selectivity |
|---|---|
| ^[bcd]\S+ | 16.1% |
| ^[bc]\S+ | 12.0% |
| ^[b]\S+ | 4.6% |

**Table 1: Observed selectivities for various regular expressions in the 100GB corpus of text.**

Each type of optimization requires dedicated support in this step. In the case of the selection optimization, the **execution fabric** needs B+Tree support in order to carry out $X$. In the event that no interesting optimization is possible from the **optimizer** step above, the **execution fabric** simply reverts to standard MapReduce execution, involving a scan of the entire contents of the input file $I$.

## 2.4 Other Optimizations

We now discuss two additional optimizations that are in the Manimal prototype; these optimizations are discovered and executed using same processing pipeline. The first is *projection*. A program that does not use 100% of its input data performs wasted work - loading, sorting, writing to network or disk - when processing these ignored bytes. For example, a program that gathers a list of high-PageRank URLs from a set of full downloaded Web page objects will use the `pageRank` and `url` portions of the data, but not `pageContent`. By using an index that contains only the data that is useful to the program, MANIMAL should yield faster runtimes. The **analyzer** can detect projections by again computing a DAG representation of *map*, examining which parts of the input `key, value` pairs are actually used in the body of the *map*. The index structure is akin to a view on the original data, with the useless attributes missing. In some ways, this optimization resembles a column-store [12].

There are several ways in which MANIMAL can improve performance by aggressively *compressing data*. First, because there is no semantic information associated with a bytestream-oriented file, any compression in conventional MapReduce must be applied to the total set of bytes. By analyzing how a MapReduce program deserializes its input, MANIMAL can attempt a data-sensitive compression of the file, compressing all the ints together, all the strings, *etc.* Second, parts of a MapReduce program may be able to operate directly on compressed data - consider a program that uses URLs as keys but never emits them would be a good candidate. By detecting such programs automatically, MANIMAL can keep the data compressed and thereby avoid some decompression work. Both of these techniques have been previously known to researchers in an RDBMS setting [1].

# 3. EXPERIMENTS

We examined the concrete improvements in MapReduce execution time that MANIMAL enables. We ran a set of experiments on a cluster of 10 server-class machines. All nodes had local direct-attached storage and were interconnected using Gigabit Ethernet. Data was stored using HDFS. We compared MANIMAL with Hadoop MapReduce version 0.20.1.

The program we evaluated was the Grep regular-expression code from Section 1. We tested it using an algorithmically-generated 100GB text file as input. The file consisted of lines of text ranging from 5 to 20 words long, each word randomly drawn from a vocabulary of 1000 different English-language words. We evaluated several regular expressions that test

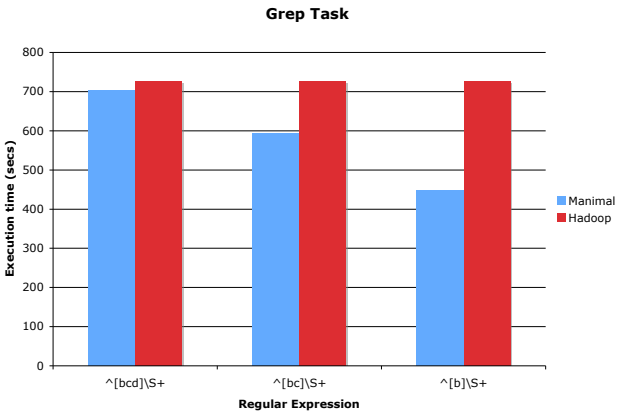


**Figure 2: Conventional Hadoop vs MANIMAL grep MapReduce program execution.**

the start of each string; although this scenario may seem obscure, it is very useful for many applications, such as testing the domain of URLs from a crawl. The specific regular expressions we tested are listed in Table 3, along with the selectivity of each. We ran each regular expression 5 times on Hadoop and 5 times using MANIMAL, dropping the fastest and slowest times, and averaging the remaining 3 runs. Distributed construction of the B+Tree index took 26.5 minutes.

Figure 2 shows the results. As expected, Hadoop's conventional MapReduce execution time is almost wholly insensitive to the selectivity of the program. MANIMAL execution time, in contrast, decreases as the conditional test becomes more restrictive, dropping to 63% of Hadoop in the case of `^[b]\S+`. There is nothing about the MANIMAL approach *per se* that dictates the selectivity level at which index-based selection optimization should be useful - we are simply applying a well-known technique to solving the problem. We believe that MANIMAL runtimes could be improved further, if it were not for relatively high job startup costs and the correct but unoptimized code that performs MANIMAL's per-tuple selection processing.

# 4. FUTURE OPTIMIZATIONS

There are several other MapReduce programming idioms we want to detect and optimize, some of which have direct analogues to optimization in relational databases. We have already mentioned column-oriented and data-compression approaches in Section 2.4. Another is conditional emission of tuples in the *reduce* function, generally applied to an aggregate of the map outputs. This is akin to a `HAVING` clause in a relational `GROUPBY` operation. If the condition is highly selective, it suggests that the program's *map* function emits a large number of tuples that are never reflected in the program's output. If we can detect and eliminate these "wasted" tuples before *reduce* processing (even imperfectly), MANIMAL could avoid a substantial amount of work.

We also plan to apply MANIMAL to non-performance-related optimization tasks. For example, one version of MapReduce execution may have excellent recovery properties that can only be obtained at some fixed performance cost, while another version may have neither recovery properties nor a fixed associated cost. Depending on the state of the cluster and the current job, MANIMAL should be able to choose between these plans. By enabling relational-style optimization, MANIMAL should allow us to make much more interesting tradeoffs for MapReduce execution than have been previously considered.

# 5. CONCLUSIONS

We demonstrated MANIMAL, a framework for applying relational optimizations to MapReduce programs. We also described a *safe* and *effective* technique for detecting selection optimizations. Finally, we showed large performance gains on a real-life MapReduce program, running in just 63% of the time otherwise required. Overall, we believe that MANIMAL points the way toward a much more sophisticated model of MapReduce program execution, applying decades of database research that have until now been deployed primarily in RDBMSes.

*Acknowledgments.* This work was partially supported by NSF Grant CRI-0707437. We are grateful to Spyros Blanas, Daniel Fabbri, H.V. Jagadish, Kristen LeFevre, and Arnab Nandi for helpful feedback.

# 6. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD Conference*, pages 671–682, 2006.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[3] F. Afrati and J. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, 2010.

[4] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools: Second Edition*. Addison-Wesley, 2007.

[5] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *ICDE*, pages 419–430, 2002.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] B. Hindman, A. Konwinski, M. Zaharia, and I. Stoica. A Common Substrate for Cluster Computing. In *HotCloud*, June 2009.

[8] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, pages 261–276, 2009.

[9] Mahout. `http://lucene.apache.org/mahout/index.html`, 2009.

[10] 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, 2008.

[11] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[12] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-Oriented DBMS. In *VLDB*, pages 553–564, 2005.

[13] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD Conference*, pages 1029–1040, 2007.

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, pages 29–42, 2008.