Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop

Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu
{mar, pkoutris, billhowe, suciu}@cs.washington.edu

University of Washington

Abstract. We explore the design and implementation of a scalable Datalog system using Hadoop as the underlying runtime system. Observing that several successful projects provide a relational algebra-based programming interface to Hadoop, we argue that a natural extension is to add recursion to support scalable social network analysis, internet traffic analysis, and general graph query. We implement semi-naive evaluation in Hadoop, then apply a series of optimizations spanning fundamental changes to the Hadoop infrastructure to basic configuration guidelines that collectively offer a 10x improvement in our experiments. This work lays the foundation for a more comprehensive cost-based algebraic optimization framework for parallel recursive Datalog queries.

1 Introduction

The MapReduce programming model has had a transformative impact on data-intensive computing, enabling a single programmer to harness hundreds or thousands of computers for a single task, often after only a few hours of development. When processing with thousands of computers, a different set of design considerations can dominate: I/O scalability, fault tolerance, and programming flexibility. The MapReduce model itself, and especially the open source implementation Hadoop [11], have become very successful by optimizing for these considerations.

A critical success factor for MapReduce has been its ability to turn a "mere mortal" java programmer into a distributed systems programmer. It raised the level of abstraction for parallel, highly scalable data-oriented programming. But it did not raise the level of abstraction high enough, evidently, because some of the earliest and most successful projects in the Hadoop ecosystem provided declarative languages on top of Hadoop. For example, HIVE provided an SQL interface, and Yahoo's Pig provided a language that closely resembles the relational algebra[1].

MapReduce (as implemented in Hadoop) has proven successful as a common runtime for non-recursive relational algebra-based languages. Our thesis is

[1] Relational algebra is not traditionally considered declarative, but Pig programs, while syntactically imperative, can be optimized by the system prior to execution and generally provide a significantly higher level of abstraction than MapReduce, so we consider the term applicable.

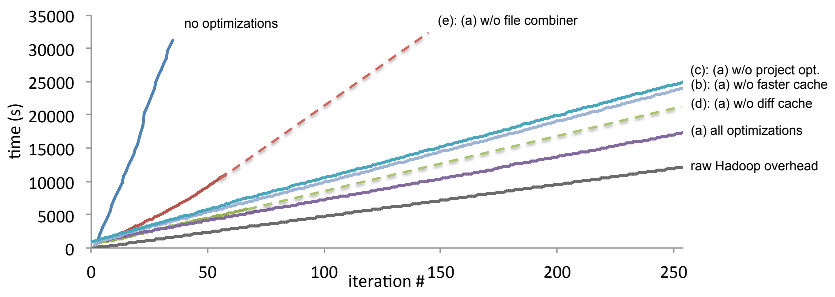

Fig. 1. The cumulative effect of our optimizations on overall runtime. (a) All optimizations applied. (b) Relaxing the specialization of the cache for inner joins. (c) Relaxing the optimization to eliminate the project operator. (d) Relaxing the diff cache optimization (extrapolated).

that Hadoop, suitably extended, can also be successful as a common runtime for recursive languages as required for graph analytics [18, 7], AI and planning applications [9], networking [14]. In previous work, our group extended Hadoop with caching and scheduling features to avoid reprocessing loop-invariant data on every iteration and to afford expression of multi-step loop bodies and various termination conditions. In this paper, we describe how this underlying framework, appropriately extended and configured, can be used to implement a scalable Datalog engine.

In this paper we present some engineering solutions for the semi-naive evaluation of a linear Datalog on Hadoop-based systems. The input to our system is a Datalog query. The query is parsed, optimized, and compiled into a series of MapReduce jobs, using an extended implemenation that directly supports iteration [7]. Our semi-naive algorithm requires three relational operators: join, duplicate elimination, and set difference, and each of them requires a separate MR job. We consider a series of optimizations to improve on this basic strategy.

Figure 1 summarizes the cumulative effect of a series of optimizations designed to improve on this basic strategy using a simple reachability query as a test case. The dataset is a social network graph dataset with 1.4 billion unique edges. The x-axis is the iteration number and the y-axis is the cumulative runtime: the highest point reached by a line indicates the total runtime of the job. Dashed lines indicate extrapolation from incomplete experiments. Figure 1(a) gives the runtime when all optimizations are applied, which is 10X-13X faster than Hadoop itself (labeled no optimizations in the figure), and only about 40% higher than the raw Hadoop overhead required to run two no-op jobs with degenerate Map and Reduce functions (labeled raw Hadoop overhead in the figure.)

The optimizations are as follows. First, for join, we notice that one of the relations in the join is loop invariant: using a previously developed cache for Hadoop [7], we store this invariant relation at the reducers, thus avoiding the expensive step of scanning and shuffling this relation at every datalog iteration. The join cache is the most significant optimization; all experiments (a)-(e) in

Figure 1 use some form of join cache. Second, by specializing and indexing the cache to implement inner joins, we can avoid many unnecessary reduce calls required by the original MapReduce semantics (Figure 1(b) show the effect of relaxing this optimization). Third, we notice that the duplicate elimination and difference can be folded into a single MR job: the newly generated tuples are shuffled to the reducers, and the same reduce job both eliminates duplicates, and checks if these tuples have already been discovered at previous iterations (Figure 1(c) shows the effect of relaxing this optimization). Fourth, with appropriate extensions to support cache insertions during execution, we observe that the cache framework can also be used to improve performance of the set difference operator (Figure 1(d) shows the effect of relaxing this optimization). Finally, we found it necessary to combine files after every job to minimize the number of map tasks in the subsequent job (Figure 1(e) shows the effect of relaxing this optimization).

In Section 3, we describe the implementation of semi-naive evaluation and the optimizations we have applied. In Section 5, we analyze these optimizations to understand their relative importance.

2 Related Work

This work is based on the MapReduce framework, which was first introduced in [8]. In this project we build our system using the popular open source implementation of MapReduce, Hadoop.

There has been an extensive line of research on providing a higher level interface to the MapReduce programming model and its implementation in Hadoop, including Dryad [12], DryadLINQ, Hyracks [6], Boom [3] and PigLatin [16]. Of these, only Hyracks provides some support for iterative queries, and they do not expose a Datalog programming interface and do not explore logical optimizations for iterative programs.

Parallel evaluation of logic systems including Datalog has been studied extensively. Balduccini et al. explore vertical (data) and horizontal (rules) parallelism, but evaluate their techniques only on small, artificial datasets [4]. Perri et al. consider parallelism at three levels: components (strata), rules, and within a single rule and show how to exploit these opportunities using modern SMP and multicore machines [17]. We target several orders of magnitude larger datasets (billions of nodes rather than tens of thousands) and have a very general model of parallelism that subsumes all three levels explore by Perri.

Damásio and Ferreira consider algorithms for improving transitive closure on large data, seeking to reduce the number of operations and iterations required. They apply these techniques to semantic web datasets and evaluate them on a single-site PostgreSQL system. The datasets contain up to millions of nodes, still three orders of magnitude smaller than our target applications.

In addition to our work on HaLoop [7], there have been several other systems providing iterative capabilities over a parallel data flow system. Pregel [15], a system developed for graph processing using the BSP model, also supports recursive operations. The Twister system [10] retains a MapReduce programming model, but uses a pub-sub system as the runtime to make better use of main memory and improve performance. The Spark system [19] also makes better use of main memory, but introduces a relational algebra-like programming model along with basic loop constructs to control iteration. The Piccolo project supports a message-passing programming model with global synchronization barriers. The programmer can provide locality hints to ensure multiple tables are co-partitioned. The Daytona project [5] at Microsoft Research provides iterative capabilities over a MapReduce system implemented on the Azure cloud computing platform. These projects all assume an imperative programming model. We explore a declarative programming model to reduce effort and expose automatic optimization opportunities. The BOOM project [3] is a distributed datalog system that provides extensions for temporal logic and distributed protocols. All of these systems are potential runtimes for Datalog. Some, but not all, of our optimizations will be applicable in these contexts as well.

A recent line of research examines recursive computations on MapReduce theoretically. In this work [2], the authors discuss issues, problems and solutions about an implementation of Datalog on the MapReduce framework. On a continuation of this work, the authors find a class of Datalog queries where it is possible to drastically reduce (to a logarithmic number) the number of recursion steps without significantly increasing the communication/replication cost.

3 Optimizing Semi-Naive Evaluation in Hadoop

Our basic execution model for Datalog in Hadoop is semi-naive evaluation. For illustration, consider a simple reachability query:

```
A(x,y) :- R(x,y), x=1234
A(x,y) :- A(x,z), R(z,y)
```

A (fully) naïve execution plan MapReduce is intuitively very expensive. On each iteration, the naïve plan requires one MR job for the join to find the next generation of results, a second job to project the result of the join and remove duplicate answers, a third MR job to compute the union (with duplicate elimination) with results discovered in previous iterations, and a fourth MR job to test for fixpoint. The inputs to each of these jobs are potentially large, distributed datasets.

An improvement is semi-naive evaluation, captured as follows.

```
$\Delta A^0 = \sigma_{x=1234}(R), i = 1$
while $\Delta A^{i-1}$ is not empty:
  $A^i = (\Delta A^0 \cup \cdots \cup \Delta A^{i-1})$
  $\Delta A^i = \pi_{xy}(\Delta A^{i-1} \bowtie_{z=z} R) - A^i, \quad i \leftarrow i+1$
```

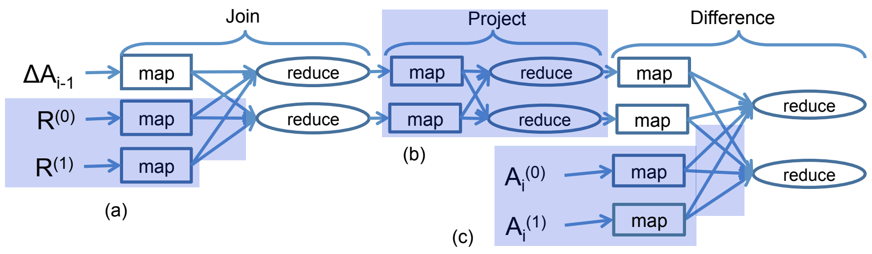

Fig. 2. Semi-naive evaluation implemented in Hadoop. (a) R is loop invariant, but gets loaded and shuffled on each iteration. (b) The extra mapreduce step to implement the project operator can be avoided by extending the join and difference operators appropriately. (c) A_i grows slowly and monotonically, but is loaded and shuffled on each iteration.

The final result is the concatenation of the results of all previous iterations ΔA^i. There is no need to remove duplicates.

In Hadoop, this execution plan involves only three MR jobs: one for the join $\bowtie_{z=z}$, one to compute the projection π_{xy} and remove duplicates, and one to compute the difference of the new results and all previous results. Computing the union of all previous results does *not* require an independent MR job, and in fact requires no work at all. The reason is that the input to a MR job is a set of files that are logically concatenated, which is just what we need.

Figure 2 illustrates the optimization opportunities for each operator. At (a), the EDB R is scanned and shuffled on every iteration even though it never changes. At (b), the overhead of an extra MapReduce step to implement the project operator can be folding duplicate elimination into the difference operator. At (c), the result relation A grows slowly and monotonically, but is scanned and shuffled on every iteration. In the remainder of this section, we describe how to optimize these operations.

3.1 Join

The join $\Delta A^{i-1} \bowtie_{z=z} R$ is implemented as a *reduce-side join* as is typical in Hadoop. The map phase hashes the tuples of both relations by the join key (and optionally applies a selection condition if the query calls for it). The reduce phase then computes the join for each unique key (the cross product of $\sigma_{z=k}\Delta A^{i-1}$ and $\sigma_{z=k}R$ for each key k). The join uses Hadoop's secondary sort capabilities to ensure that only $\sigma_{z=k}\Delta A^{i-1}$ incurs memory overhead; $\sigma_{z=k}R$ can be pipelined. Skew issues can be addressed by using an out-of-core algorithm such as hybrid hash join or by other common techniques.

The critical bottleneck with the join operation is that the entire relation R must be scanned and shuffled on each and every iteration. In previous work on HaLoop [7], we added a *Reducer Input Cache* (RIC) to avoid scanning and shuffling loop-invariant relations (problem (a)). Specifically, HaLoop will cache the reducer inputs across all reduce nodes and create an index for the cached data and stores it on local disk. Reducer inputs are cached during reduce function invocation, so the tuples in the reducer input cache are sorted and grouped by reducer input key. When a reducer processes a shuffled key and its values, it searches the appropriate local reducer input cache to find corresponding keys and values. An iterator that combines the shuffled data and the cached data is then passed to the user-defined reduce function for processing. In the physical layout of this cache, keys and values are separated into two files, and each key has an associated pointer to its corresponding values. Since the cache is sorted and accessed in sorted order, only one sequential scan must be made in the worst case. Fault-tolerance was preserved by arranging for caches to be rebuilt when failures occurred without having to rerun all previous iterations.

To be fully transparent with the original MapReduce semantics, all cached keys, regardless of whether they appear in the mapper or not, should be passed to the reducer for processing. However, the equijoin semantics used in Datalog expose an optimization opportunity: Only those cached values that match a value in the incoming mapper output need be extracted and passed to the reducer, for significant savings (Figure 1(b)).

3.2 Difference

In Figure 2(b), the set difference operator compares the generated join output to the loop's accumulated result to ensure that only the newly discovered tuples are output in the operation's result. By default, the difference operation requires the scanning and shuffling of all previous iterations' results on every iteration. As the number of iterations increases, both the number of (possibly small) files accessed and the amount of data being reshuffled increases. Each file requires a separate map task, so it is important to group the many small files before computing the difference. The performance improvement of this configuration detail is significant (Figure 2(d)).

Like the join operation, the difference operation in Figure 2(b) can benefit from a cache. The previously discovered results need not be scanned and shuffled on each iteration; instead, these values can be maintained in a cache and updated on each iteration. We extended the original HaLoop caching framework to support insertions during iterative processing, generalizing it for use with the difference operator.

The difference operator uses the cache as follows. Each tuple is stored in the cache as a key-value pair (t, i), where the key is the tuple t discovered by the previous join operator and the value is the iteration number i for which that tuple was discovered. On each iteration, the map phase of the difference operator hashes the incoming tuples as keys with values indicating the current iteration number. During the reduce phase, for each incoming tuple (from the map phase), the cache is probed to find all instances of the tuples previously discovered across all iterations. Both the incoming and cached data are passed to the user-defined reduce function. Any tuples that were previously discovered are suppressed in the output.

For example, consider a reachability query. If a node t was discovered on iteration 1, 4, 5, and 8, there would be four instances of t in the cache when the reduce phase of iteration 8 executes. The reducer then would receive a list of five key-value pairs: the pair $(t, 8)$ from the map phase, and the pairs $(t, 1)$, $(t, 4)$, $(t, 5)$, $(t, 8)$ from the cache. The reducer can then determine that the tuple had been seen before, and therefore emit nothing. If the tuple had never before been seen, then it would receive a singleton list (t,8) and would recognize that this tuple should be included in ΔA^9 and emit the tuple.

To avoid storing duplicate values in the cache, we leverage Haloop's cache-filtering functionality. When a reduce operation processes an individual value associated with a key, Haloop invokes the user-defined $isCache()$ routine to first consult a hashtable of all tuples previously written to the cache during this iteration. We reduce the number of duplicates that must be considered by this mechanism by using a combiner function on the map side. Combiners are a common Hadoop idiom used to reduce the amount of communication between mappers and reducers by pre-aggregating values. In this case, the combiner ensures that each mapper only produces each value once.

A limitation in the current Haloop cache implementation is that the cache is rewritten completely on every iteration during which new tuples are discovered, incurring significant IO overhead in some cases. A redesigned cache that avoids this step is straightforward, but remains future work.

3.3 Project

The project operator at Figure 2(b) is implemented as a separate MapReduce job. The two tasks accomplished in this job — column elimination and duplicate row elimination — can be delegated to the join operator and the difference operator, respectively.

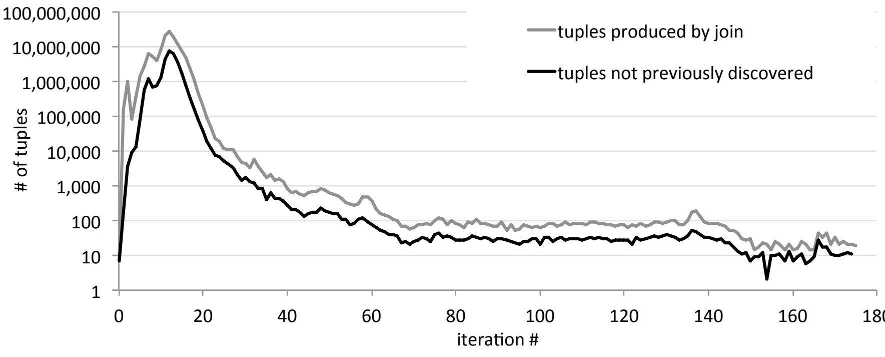

Fig. 3. The "endgame" of a recursive computation. The number of new tuples discovered peaks early (y-axis, log scale). Beyond iteration 25, less than 1000 new nodes are discovered on each iteration though execution continues for hundreds of iterations.

The difference operator naturally removes duplicates as a side effect of processing. To remove columns, we have made a straightforward extension to the join operator to provide a *column-selecting join* that is capable of removing columns. Since it does not need to remove duplicates, this step can be performed entirely in parallel and therefore incurs very little overhead.

By replacing sequences of Join→Project→Difference operations with sequences of ColumnSelectingJoin→Difference, the system is able to eliminate a map-reduce job *per iteration* for significant savings (Figure 1(c)).

4 Implementation

We have implemented a Datalog interpreter that converts schema and rules to a sequence of MapReduce jobs. The Datalog query is converted into the relational algebra, optimized (if desired), and then translated into a set of MapReduce jobs that are configured through JobConf configuration parameters. A sample input to the interpreter is seen below:

```
1) backend R[long, long, long] "btc".
2) res(x) :- R(1570650593L, b, x) .
3) res(y) :- res(z), R(z,b,y) .
4) ans res(y)
```

Line (1) specifies the schema for the R dataset, which consists of three columns of type long, and backed by the Hadoop directory "btc." Input directories can contain either delimited text files or SequenceFiles. Lines (2–3) define rules available for evaluating the query answer, specified on line (4).

5 Analysis and Evaluation

We evaluate our datalog system on the 2010 Billion Triple Challenge (BTC) dataset, a large graph dataset (625GB uncompressed) with 2B nodes and 3.2B quads. Queries are executed on a local 21-node cluster, consisting of 1 master node and 20 slave nodes. The cluster consists of Dual Quad Core 2.66GHz and 2.00GHz machines with 16GB RAM, all running 64bit RedHat Enterprise Linux. Individual MapReduce jobs are given a maximum Java heap space of 1GB.

The source of the BTC data is a web crawl of semantic data, and the majority of the nodes are associated with social network information. The graph is disconnected, but Joslyn et al [13] found that the largest component accounts for 99.8% of the distinct vertices in the graph.

As a result of these properties, recursive queries over this graph are challenging to optimize. In particular, they exhibit the endgame problem articulated by Ullman et al [2]. To illustrate the problem, consider Figure 3 which shows the number of nodes encountered by iteration number for a simple reachability query (i.e., find all nodes connected to a particular node.) The x-axis is the iteration number and the y-axis is the number of new nodes discovered in log-scale. The two datasets represent the size of the results of the join and difference operators. The nodes encountered by the join operation are in red and the nodes that are determined to not have not been previously discovered are in blue. The overall shape of the curve is striking. In iteration 17, over 10 million new nodes are discovered as the frontier passes through a well-connected region of the graph. By iteration 21, however, the number of new nodes produced has dropped precipitously. In ongoing work, we are exploring dynamic reoptimization techniques to make both regimes efficient. In this paper, we focus on the core hadoop-related system optimizations required to lay a foundation for that work.

In this evaluation, we consider the following questions:

– Join Cache: How much improvement can we expect from the reducer input cache in the context of long-running recursive queries and semi-naive evaluation?
– Diff Cache: How much improvement can we expect from an extended cache subsystem suitable for use with the difference step of semi-naive evaluation?
– Equijoin semantics: If we optimize for equi-joins in the underlying subsystem, how much improvement can we expect over the original MapReduce semantics where every key from both relations must be processed?

To answer these questions, we consider a simple reachability query

```
A(y) :- R(startnode,y)
A(y) :- A(x),R(x,y)
```

where *startnode* is a literal value that refers to a single node id. This query returns 29 million nodes in the result and runs for 254 iterations for the *startnode* we have selected. The query is simple, but clearly demonstrates the challenges of optimizing recursive queries in this context.

Unless otherwise noted in the text, the individual identifiers in the BTC2010 data set are hashed to long integer values and stored as SequenceFiles.

After 33 iterations, our cumulative running time for (2) Join → Proj → Diff is less than 15% of the query time when no caches are used. Additionally, modifying our operators to eliminate an unnecessary Hadoop job to implement the Project operator reduces iteration time by more than 26 seconds per iteration. Over the course of 254 iterations, this adds up to over 100 minutes of query time. After 33 iterations, the cumulative query time for (3) ColumnSelectingJoin → Diff is less than 10% of the cacheless query's execution time.

Figure 4 presents the per-operation contribution of the cumulative iteration time for iterations 1-33. The three scenarios are (1) ColumnSelectingJoin → Diff with no caches, (2) Join → Proj → Diff with both a join cache and a diff cache, (3) ColumnSelectingJoin → Diff with both a join cache and a diff cache, and (4) and estimate of the minimum job overhead for running two no-op Hadoop jobs on every iteration. With no caches used, the time is dominated by the join operation. With optimizations applied, the time approaches the minimum overhead of Hadoop (about 18 seconds per job for our cluster and current configuration).

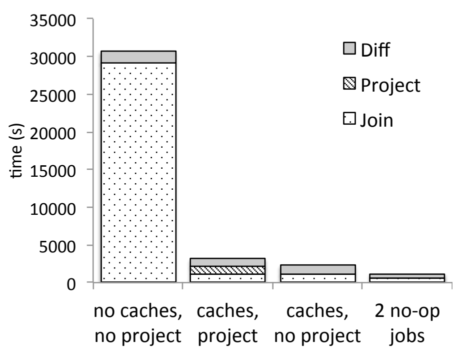

Fig. 4. The total time spent in each operation across iterations 1-33 (excluding 0) for three evaluation strategies. Without caching, the join operation dominates execution time.

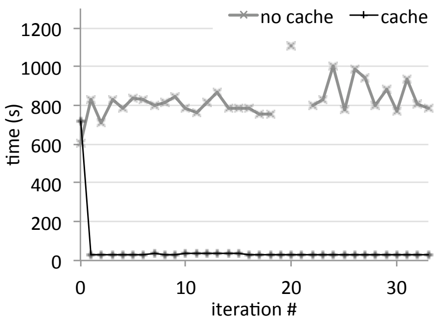

Fig. 5. Time to execute the join only. The use of a cache to avoid re-scanning and reshuffling the loop-invariant data on each iteration results in significant improvement (and also appears to decrease variability). Gaps represent failed jobs.

Figure 5 shows the runtime of the join step only by iteration, both with and without the cache enabled. On the first iteration, populating the cache incurs some overhead. (We plot successful jobs only for clarity; gaps in the data represent inaccurate runtime measurements due to job failures.) On subsequent iterations, the invariant graph relation need not be scanned and shuffled, resulting in considerable savings. For our test query, the time for the join per iteration went from approximately 700 seconds to approximately 30 seconds, and the overall runtime of the query went from 50 hours to 5 hours (making these kind of jobs feasible!)

In Figure 6, the cache subsystem has been extended to allow new nodes to be added to it on each iteration. The y-axis shows the runtime of the difference operator, and the x-axis is the iteration number. This capability was not available in HaLoop and is used to improve semi-naive evaluation. Without the cache, each iteration must scan more and more data, resulting in a (slow) increase in the per-iteration runtime. With the cache, the increase in size has no discernible effect on performance, and the performance of the operator improves by about 20%. The outlier values were the result of failures. The fact that no failures occurred when using the cache is not statistically significant.

The cache is specialized for the equijoin case, improving performance over the original MapReduce semantics. Specifically, the original semantics dictates that the reduce function must be called for every unique key, including all those keys in the cache that do not have a corresponding joining tuple in ΔA_i during semi-naive evaluation. By exposing a datalog interface rather than raw MapReduce, we free ourselves from these semantics and can only call the reduce function once for each incoming key. In Figure 7, the effect of this specialization is measured for the first few iterations.

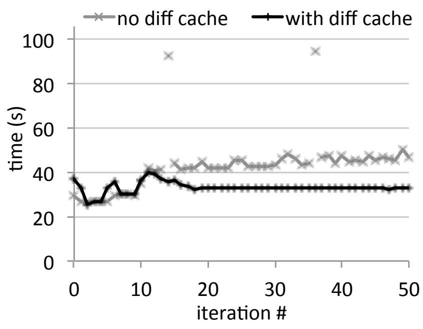

Fig. 6. Time per iteration for the difference operator only. The cache improves performance by about 20%.

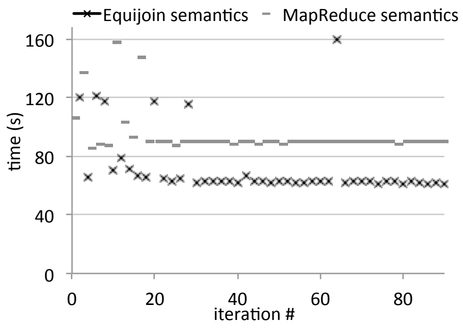

Fig. 7. The per-iteration impact of specializing the cache for equijoin, which is safe given our target lanaguage of datalog.

6 Conclusions and Future Work

Informed by Hadoop's success as a runtime for relational algebra-based languages, and bulding on our previous work on the HaLoop system for iterative processing, we explore the suitability of Hadoop as a runtime for recursive Datalog. We find that caching loop invariant data delivers an order of magnitude speedup, while specialized implementations of the operators, careful configuration of Hadoop for iterative queries, and extensions to the cache to support set difference delivers another factor of 2 speedup.

We also find that the overhead of an individual Hadoop job is significant in this context, as it amplified by the iterative processing (500+ Hadoop jobs are executed to evaluate one query!) This overhead accounts for approximately half of time of each iteration step after all optimizations are applied.

In future work on system optimizations, we are considering extensions to Hadoop to optimize the caching mechanism and avoid unnecessary IO by using a full-featured disk-based indexing libraryon each node in a manner similar to HadoopDB [1]. We are also exploring the literature for new ways to mitigate the startup overheaad of each Hadoop job by sharing VMs across jobs.

Perhaps more importantly, we are aggressively exploring cost-based algebraic dynamic re-optimization techniques for parallel recursive queries, motivated in particular by the endgame problem (Figure 3). While this paper explored Hadoop in particular, our ongoing work is designed to produce optimization strategies that will transcend any particular distributed runtime. To demonstrate this, we are planning experiments that use more recent parallel runtimes that directly support recursion [10, 19].

References

1. A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for ana-

lytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, August 2009.
2. F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 1–8, New York, NY, USA, 2011. ACM.
3. P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In C. Morin and G. Muller, editors, *EuroSys*, pages 223–236. ACM, 2010.
4. M. Balduccini, E. Pontelli, O. Elkhatib, and H. Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Comput.*, 31(6):608–647, June 2005.
5. R. Barga, J. Ekanayake, J. Jackson, and W. Lu. Daytona: Iterative mapreduce on windows azure. `http://research.microsoft.com/en-us/projects/daytona/`.
6. V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE*, pages 1151–1162. IEEE Computer Society, 2011.
7. Y. Bu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proc. of International Conf. on Very Large Databases (VLDB)*, 2010.
8. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
9. J. Eisner and N. W. Filardo. Dyna: Extending datalog for modern ai. In *Datalog*, pages 181–220, 2010.
10. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC*, pages 810–818, 2010.
11. Hadoop. `http://hadoop.apache.org/`.
12. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In P. Ferreira, T. R. Gross, and L. Veiga, editors, *EuroSys*, pages 59–72. ACM, 2007.
13. C. Joslyn, R. Adolf, S. al Saffar, J. Feo, E. Goodman, D. Haglin, G. Mackey, and D. Mizell. High performance semantic factoring of giga-scale semantic graph databases. Semantic Web Challenge Billion Triple Challenge 2010.
14. B. T. Loo, H. Gill, C. Liu, Y. Mao, W. R. Marczak, M. Sherr, A. Wang, and W. Zhou. Recent advances in declarative networking. In *PADL*, pages 1–16, 2012.
15. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *SIGMOD Conference*, pages 135–146. ACM, 2010.
16. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 1099–1110. ACM, 2008.
17. S. Perri, F. Ricca, and M. Sirianni. A parallel asp instantiator based on dlv. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, DAMP '10, pages 73–82, New York, NY, USA, 2010. ACM.
18. M. Shaw, L. Detwiler, N. Noy, J. Brinkley, and S. D. vsparql: A view definition language for the semantic web. *Journal of Biomedical Informatics, doi:10.1016/j.jbi.2010.08.008.*
19. M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.