BOOM: Data-Centric Programming in the Datacenter

Peter Alvaro
Tyson Condie
Khaled Elmeleegy
Joseph M. Hellerstein
Russell C Sears

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-111
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-111.html

August 10, 2009

Copyright 2009, by the author(s).
All rights reserved.

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.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant Nos. 0722077 and 0713661, the University of California MICRO program, and gifts from Sun Microsystems, Inc. and Microsoft Corporation.

BOOM: Data-Centric Programming in the Datacenter

Peter Alvaro
UC Berkeley

Tyson Condie
UC Berkeley

Neil Conway
UC Berkeley

Khaled Elmeleegy
Yahoo! Research

Joseph M. Hellerstein
UC Berkeley

Russell Sears
UC Berkeley

ABSTRACT

Cloud computing makes clusters a commodity, creating the potential for a wide range of programmers to develop new scalable services. However, current cloud platforms do little to simplify truly distributed systems development. In this paper, we explore the use of a declarative, data-centric programming model to achieve this simplicity. We describe our experience using Overlog and Java to implement a "Big Data" analytics stack that is API-compatible with Hadoop and HDFS, with equivalent performance. We extended the system with complex features not yet available in Hadoop, including availability, scalability, and unique monitoring and debugging facilities. We present our experience to validate the enhanced programmer productivity afforded by declarative programming, and to inform the design of new development environments for distributed programming.

1. INTRODUCTION

Clusters of commodity hardware have become a standard architecture for datacenters over the last decade, and the major online services have all invested heavily in cluster software infrastructure (e.g., [4, 5, 9, 10, 12]). This infrastructure consists of distributed software that manages difficult issues including parallelism, communication, failure, and system resizing. Such systems support basic service operation, and facilitate software development by in-house developers.

The advent of *cloud computing* promises to commoditize datacenters by making it simple and economical for third-party developers to host their applications on managed clusters. Unfortunately, writing distributed software remains as challenging as ever, which impedes the full utilization of the power of this new platform. The right programming model for the cloud remains an open question.

Current cloud platforms provide conservative, "virtualized legacy" interfaces to developers, which typically take the form of traditional single-node programming interfaces in an environment of hosted virtual machines and shared storage. For example, Amazon EC2 exposes "raw" VMs and distributed storage as their development environment, while Google App Engine and Microsoft Azure provide programmers with traditional single-node programming languages and APIs to distributed storage. These single-node models were likely chosen for their maturity and familiarity, rather than their ability to empower developers to write innovative distributed programs.

A notable counter-example to this phenomenon is the MapReduce framework popularized by Google [9] and Hadoop, which has enabled a wide range of developers to easily coordinate large numbers of machines. MapReduce raises the programming abstraction from a traditional von Neumann model to a functional dataflow model that can be easily auto-parallelized over a shared-storage architecture. MapReduce programmers think in a *data-centric* fashion: they worry about handling sets of data records, rather than managing fine-grained threads, processes, communication and coordination. MapReduce achieves its simplicity in part by constraining its usage to batch-processing tasks. Although limited in this sense, it points suggestively toward more attractive programming models for datacenters.

1.1 Data-centric programming in BOOM

Over the last twelve months we have been working on the *BOOM* project, an exploration in using data-centric programming to develop production-quality datacenter software.[1] Reviewing some of the initial datacenter infrastructure efforts in the literature (e.g., [5, 12, 10, 9]), it seemed to us that most of the non-trivial logic involves managing various forms of asynchronously-updated state — sessions, protocols, storage — rather than intricate, uninterrupted sequences of computational steps. We speculated that the Overlog language used for Declarative Networking [18] would be well-suited to those tasks, and could significantly ease the development of datacenter software without introducing major computational bottlenecks. The initial *P2* implementation of Overlog [18] is aging and targeted at network protocols, so we developed a new Java-based Overlog runtime we call *JOL* (Section 2).

To evaluate the feasibility of BOOM, we chose to build *BOOM Analytics*: an API-compliant reimplementation of the Hadoop MapReduce engine and its HDFS distributed file system. In writing BOOM Analytics, we preserved the Java API "skin" of Hadoop and HDFS, but replaced their complex internals with Overlog. The Hadoop stack appealed to us for two reasons. First, it exercises the distributed power of a cluster. Unlike a farm of independent web service instances, the Hadoop and HDFS code entails coordination of large numbers of nodes toward common tasks. Second, Hadoop is a work in progress, still missing significant distributed features like availability and scalability of master nodes. The difficulty of adding these complex features could serve as a litmus test of the

[1] BOOM stands for the *Berkeley Orders Of Magnitude* project, which aims to build orders of magnitude bigger systems in orders of magnitude less code.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '09, August 24-28, 2009, Lyon, France
Copyright 2009 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

programmability of our approach.

1.2 Contributions

The bulk of this paper describes our experience implementing and evolving BOOM Analytics, and running it on Amazon EC2. After twelve months of development, BOOM Analytics performed as well as vanilla Hadoop, and enabled us to easily develop complex new features including Paxos-supported replicated-master availability, and multi-master state-partitioned scalability. We describe how a data-centric programming style facilitated debugging of tricky protocols, and how by metaprogramming Overlog we were able to easily instrument our distributed system at runtime. Our experience implementing BOOM Analytics in Overlog was gratifying both in its relative ease, and in the lessons learned along the way: lessons in how to quickly prototype and debug distributed software, and in understanding limitations of Overlog that may contribute to an even better programming environment for datacenter development.

This paper presents the evolution of BOOM Analytics from a straightforward reimplementation of Hadoop/HDFS to a significantly enhanced system. We describe how an initial prototype went through a series of major revisions ("revs") focused on *availability* (Section 4), *scalability* (Section 5), and *debugging and monitoring* (Section 6). In each case, the modifications involved were both simple and well-isolated from the earlier revisions. In each section we reflect on the ways that the use of a high-level, data-centric language affected our design process.

1.3 Related Work

Declarative and data-centric languages have traditionally been considered useful in very few domains, but things have changed substantially in recent years. MapReduce [9] has popularized functional dataflow programming with new audiences in computing. And a surprising breadth of research projects have proposed and prototyped declarative languages in recent years, including overlay networks [18], three-tier web services [30], natural language processing [11], modular robotics [2], video games [29], file system metadata analysis [13], and compiler analysis [15].

Most of the languages cited above are declarative in the same sense as SQL: they are based in first-order logic. Some — notably MapReduce, but also SGL [29] — are algebraic or dataflow languages, used to describe the composition of operators that produce and consume sets or streams of data. Although arguably imperative, they are far closer to logic languages than to traditional imperative languages like Java or C, and are often amenable to set-oriented optimization techniques developed for declarative languages [29]. Declarative and dataflow languages can also share the same runtime, as demonstrated by recent integrations of MapReduce and SQL in Hive [27], DryadLINQ [31], HadoopDB [1], and products from vendors such as Greenplum and Aster.

Concurrent with our work, the Erlang language was used to implement a simple MapReduce framework called Disco [22], and a transactional DHT called Scalaris with Paxos support [23]. Philosophically, Erlang revolves around programming *concurrent processes*, rather than data. In Section 7 we reflect on some benefits of a data-centric language over Erlang's approach. For the moment, Disco is significantly less functional than BOOM Analytics, lacking a distributed file system, multiple scheduling policies, and high availability via consensus.

Distributed state machines are the traditional formal model for distributed system implementations, and can be expressed in languages like Input/Output Automata (IOA) and the Temporal Logic of Actions (TLA) [20]. By contrast, our approach is grounded in Datalog and its extensions. The pros and cons of starting with a database foundation are a recurring theme of this paper.

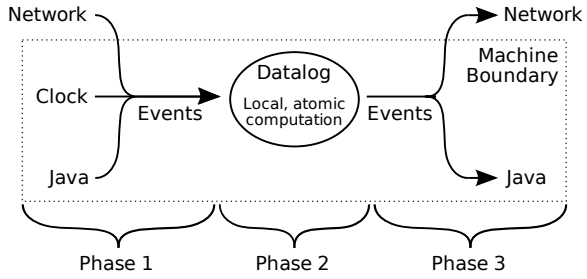

Figure 1: An Overlog timestep at a participating node: incoming events are applied to local state, the local Datalog program is run to fixpoint, and any outgoing events are emitted.

Our use of metaprogrammed Overlog was heavily influenced by the Evita Raced Overlog metacompiler [8], and the security and typechecking features of Logic Blox' LBTrust [21]. Some of our monitoring tools were inspired by Singh et al. [25], although our metaprogrammed implementation is much simpler and more elegant than that of P2.

2. BACKGROUND

The Overlog language is sketched in a variety of papers. Originally presented as an event-driven language [18], it has evolved a more pure declarative semantics based in Datalog, the standard deductive query language from database theory [28]. Our Overlog is based on the description by Condie et al. [8]. We review Datalog in Appendix A, and the extensions offered by Overlog here.

Overlog extends Datalog in three main ways: it adds notation to specify the location of data, provides some SQL-style extensions such as primary keys and aggregation, and defines a model for processing and generating changes to tables. Overlog supports relational tables that may optionally be "horizontally" partitioned row-wise across a set of machines based on a column called the *location specifier*, which is denoted by the symbol `@`. (Appendix A shows a standard network routing example from previous papers on declarative networking.)

When Overlog tuples arrive at a node either through rule evaluation or external events, they are handled in an atomic local Datalog "timestep". Within a timestep, each node sees only locally-stored tuples. Communication between Datalog and the rest of the system (Java code, networks, and clocks) is modeled using *events* corresponding to insertions or deletions of tuples in Datalog tables.

Each timestep consists of three phases, as shown in Figure 2. In the first phase, inbound events are converted into tuple insertions and deletions on the local table partitions. In the second phase, we run the Datalog rules to a "fixpoint" in a traditional bottom-up fashion [28], recursively evaluating the rules until no new results are generated. In the third phase, updates to local state are atomically made durable, and outbound events (network messages, Java callback invocations) are emitted. Note that while Datalog is defined over static databases, the first and third phases allow Overlog programs to mutate state over time.

JOL is an Overlog runtime implemented in Java, based on a dataflow of operators similar to P2 [18]. JOL implements *metaprogramming* akin to P2's Evita Raced extension [8]: each Overlog program is compiled into a representation that is captured in rows of tables. As a result, program testing, optimization and rewriting can be written concisely in Overlog to manipulate those tables. JOL supports Java-based extensibility in the model of Postgres [26]. It supports Java classes as abstract data types, allowing Java objects

to be stored in fields of tuples, and Java methods to be invoked on those fields from Overlog. JOL also allows Java-based aggregation functions to run on sets of column values, and supports Java *table functions*: Java iterators producing tuples, which can be referenced in Overlog rules as ordinary database tables.

3. INITIAL PROTOTYPE

Our coding effort began in May, 2008, with an initial implementation of JOL. By June of 2008 we had JOL working well enough to begin running sample programs. Development of the Overlog-based version of HDFS (BOOM-FS) started in September of 2008. We began development of our Overlog-based version of MapReduce (BOOM-MR) in January, 2009, and the results we report on here are from May, 2009. In Section 7 we reflect briefly on language and runtime lessons related to JOL.

We used two different design styles in developing the two halves of BOOM Analytics. For BOOM-MR, we essentially ported much of the "interesting" material in Hadoop's MapReduce code piece-by-piece to Overlog, leaving various API routines in their original state in Java. By contrast, we began our BOOM-FS implementation as a clean-slate rewrite in Overlog. When we had a prototype file system working in an Overlog-only environment, we retrofitted the appropriate Java APIs to make it API-compliant with Hadoop.

3.1 MapReduce Port

The goal of our MapReduce port was to make it easy to evolve a non-trivial aspect of the system. MapReduce scheduling policies were one issue that had been treated in recent literature [32]. To enable credible work on MapReduce scheduling, we wanted to remain true to the basic structure of the Hadoop MapReduce codebase, so we proceeded by understanding that code, mapping its core state into a relational representation, and then writing Overlog rules to manage that state in the face of new messages delivered by the existing Java APIs. We follow that structure in our discussion.

3.1.1 Background: Hadoop MapReduce

In Hadoop MapReduce, there is a single master node called the *JobTracker*, which manages a number of worker nodes called *TaskTrackers*. A job is divided into a set of map and reduce *tasks*. The JobTracker assigns tasks to worker nodes. Each map task reads a 64MB *chunk* from the distributed file system, runs a user-defined map function, and partitions output key/value pairs into hash-buckets on local disk. The JobTracker then forms reduce tasks corresponding to each hash value, and assigns these tasks to TaskTrackers. Each reduce task fetches the corresponding hash buckets from all mappers, sorts locally by key, runs the reduce function and writes the results to the distributed file system.

Each TaskTracker has a fixed number of slots for executing tasks — two maps and two reduces by default. A heartbeat protocol between each TaskTracker and the JobTracker is used to update the JobTracker's bookkeeping of the state of running tasks, and drive the scheduling of new tasks: if the JobTracker identifies free TaskTracker slots, it will schedule further tasks on the TaskTracker. Hadoop will often schedule *speculative* tasks to reduce a job's response time by preempting "straggler" nodes [9].

3.1.2 MapReduce in Overlog

Our initial goal was to port the JobTracker code to Overlog. We began by identifying the key state maintained by the JobTracker. This state includes both data structures to track the ongoing status of the system, and transient state in the form of messages sent and received by the JobTracker. We captured this information fairly naturally in four Overlog tables, shown in Table 1.

Name	*Description*	*Relevant attributes*
job	Job definitions	jobid, priority, submit_time, status, jobConf
task	Task definitions	jobid, taskid, type, partition, status
taskAttempt	Task attempts	jobid, taskid, attemptid, progress, state, phase, tracker, input_loc, start, finish
taskTracker	TaskTracker definitions	name, hostname, state, map_count, reduce_count, max_map, max_reduce

Table 1: BOOM-MR relations defining JobTracker state.

The underlined attributes in Table 1 together make up the primary key of each relation. The *job* relation contains a single row for each job submitted to the JobTracker. In addition to some basic metadata, each job tuple contains an attribute called *jobConf* that holds a Java object constructed by legacy Hadoop code, which captures the configuration of the job. The *task* relation identifies each task within a job. The attributes of this relation identify the task type (map or reduce), the input "partition" (a chunk for map tasks, a bucket for reduce tasks), and the current running status.

A task may be attempted more than once, due to speculation or if the initial execution attempt failed. The *taskAttempt* relation maintains the state of each such attempt. In addition to a progress percentage and a state (running/completed), reduce tasks can be in any of three phases: copy, sort, or reduce. The *tracker* attribute identifies the TaskTracker that is assigned to execute the task attempt. Map tasks also need to record the location of their input chunk, which is given by *input_loc*. The *taskTracker* relation identifies each TaskTracker in the cluster with a unique name.

Overlog rules are used to update the JobTracker's tables by converting inbound messages into *job*, *taskAttempt* and *taskTracker* tuples. These rules are mostly straightforward. Scheduling decisions are encoded in the *taskAttempt* table, which assigns tasks to TaskTrackers. A scheduling policy is a set of rules that join against the *taskTracker* relation to find TaskTrackers with unassigned slots, and schedules tasks by inserting tuples into *taskAttempt*.

System	*Lines in Patch*	*Files Modified by Patch*
Hadoop	2102	17
BOOM-MR	82	2

Table 2: Modifying MapReduce schedulers with LATE.

To exercise our extensible scheduling architecture, we implemented the LATE scheduler [32], in addition to Hadoop's default scheduling policy. The LATE policy is specified in the paper via just three lines of pseudocode. Table 2 quantifies the relative complexity of the Java LATE scheduler patch against Hadoop ([17], issue HADOOP-2141) with the size of our LATE implementation in BOOM-MR. Appendix C validates the faithfulness of our implementation in practice. In sum, we were pleased to see that the BOOM approach enabled scheduler modifications that were over an order of magnitude smaller than traditional approaches.

3.1.3 Discussion

We had an initial version of BOOM-MR running after a month of development, and have continued to tune it until very recently. BOOM-MR consists of 55 Overlog rules in 396 lines of code, and 1269 lines of Java. It was based on Hadoop version 18.1; we estimate that we removed 6,573 lines from Hadoop (out of 88,864) in writing BOOM-MR. The removed code contained the core scheduling logic and the data structures that represent the components listed in Table 1. The performance of BOOM-MR is very similar to that of Hadoop MapReduce, as seen in the experiments in Appendix B.

Our experience gutting Hadoop and inserting BOOM Analytics was not always pleasant. Given that we were committed to preserving the client API, we did not take a "purist" approach and try to convert everything into tables and Overlog rules. For example, we chose not to "tableize" the JobConf object, but instead to carry it through Overlog tuples. In our Overlog rules, we pass the JobConf object into a custom Java table function that manufactures *task* tuples for the job, subject to the specifications in the JobConf.

In retrospect, it was handy to be able to draw the Java/Overlog boundaries flexibly. This allowed us to focus on porting the more interesting Hadoop logic into Overlog, while avoiding ports of relatively mechanical details. We also found that the Java/Overlog interfaces we implemented in JOL were both necessary and sufficient for our needs. We employed them all: table functions for producing tuples from Java, Java objects and methods within tuples, Java aggregation functions, and Java event listeners that listen for insertions and deletions of tuples into tables.

With respect to Overlog, we found it much simpler to extend and modify than the original Hadoop Java code, as demonstrated by our experience with LATE. We have been experimenting with new scheduling policies recently, and it has been very easy to modify existing policies and try new ones. Informally, our Overlog code seems about as simple as the task should require: the coordination of MapReduce task scheduling is not a terribly rich design space, and we feel that the simplicity of BOOM-MR is appropriate to the simplicity of the system's job.

3.2 HDFS Rewrite

The BOOM-MR logic described in the previous section is based on entirely centralized state: the only distributed aspect of the code is the implementation of message handlers. Although its metadata is still centralized, the actual data in HDFS is distributed and replicated. HDFS is loosely based on GFS [12], and is targeted at storing large files for full-scan workloads.

In HDFS, file system metadata is stored at a centralized *NameNode*, but file data is partitioned into 64MB chunks and distributed across a set of *DataNodes*. Each chunk is typically stored at three DataNodes to provide fault tolerance. DataNodes periodically send heartbeat messages to the NameNode containing the set of chunks stored at the DataNode. The NameNode caches this information. If the NameNode has not seen a heartbeat from a DataNode for a certain period of time, it assumes that the DataNode has crashed and deletes it from the cache; it will also create additional copies of the chunks stored at the crashed DataNode to ensure fault tolerance.

Clients only contact the NameNode to perform metadata operations, such as obtaining the list of chunks in a file; all data operations involve only clients and DataNodes. HDFS only supports file read and append operations — chunks cannot be modified once they have been written.

3.2.1 BOOM-FS In Overlog

In contrast to our "porting" strategy for implementing BOOM-MR, we chose to build BOOM-FS from scratch. This required us to exercise Overlog more broadly, limiting our Hadoop/Java compatibility task to implementing the HDFS client API in Java. We did this by creating a simple translation layer between Hadoop API operations and BOOM-FS protocol commands. The resulting BOOM-FS implementation works with either vanilla Hadoop MapReduce or BOOM-MR.

Like GFS, HDFS maintains a clean separation of control and data protocols: metadata operations, chunk placement and DataNode liveness are cleanly decoupled from the code that performs bulk data transfers. This made our rewriting job substantially more attractive. JOL is a relatively young runtime and is not tuned for high-bandwidth data manipulation, so we chose to implement the simple high-bandwidth data path "by hand" in Java, and used Overlog for the trickier but lower-bandwidth control path. While we initially made this decision for expediency, as we reflect in Section 7, it yielded a hybrid system that is clean and efficient.

Name	*Description*	*Relevant attributes*
file	Files	fileid, parentfileid, name, isDir
fqpath	Fully-qualified pathnames	path, fileid
fchunk	Chunks per file	chunkid, fileid
datanode	DataNode heartbeats	nodeAddr, lastHeartbeatTime
hb_chunk	Chunk heartbeats	nodeAddr, chunkid, length

Table 3: BOOM-FS relations defining file system metadata.

3.2.2 File System State

The first step of our rewrite was to represent file system metadata as a collection of relations. We then implemented file system policy by writing queries over this schema. A simplified version of the relational file system metadata in BOOM-FS is shown in Table 3.

The *file* relation contains a row for each file or directory stored in BOOM-FS. The set of chunks in a file is identified by the corresponding rows in the *fchunk* relation.[2] The *datanode* and *hb_chunk* relations contain the set of live DataNodes and the chunks stored by each DataNode, respectively. The NameNode updates these relations as new heartbeats arrive; if the NameNode does not receive a heartbeat from a DataNode within a configurable amount of time, it assumes that the DataNode has failed and removes the corresponding rows from these tables.

The NameNode must ensure that the file system metadata is durable, and restored to a consistent state after a failure. This was easy to implement using Overlog, because of the natural atomicity boundaries provided by fixpoints. We used the Stasis storage library [24] to flush durable state changes to disk as an atomic transaction at the end of each fixpoint. Since JOL allows durability to be specified on a per-table basis, the relations in Table 3 were marked durable, whereas "scratch tables" that are used to compute responses to file system requests were transient.

Since a file system is naturally hierarchical, a recursive query language like Overlog was a natural fit for expressing file system policy. For example, an attribute of the *file* table describes the parent-child relationship of files; by computing the transitive closure of this relation, we can infer the fully-qualified pathname of each file (*fqpath*). (The two Overlog rules that derive *fqpath* from *file* are listed in Figure 4 in Appendix A.) Because this information is accessed frequently, we configured the *fqpath* relation to be cached after it is computed. Overlog will automatically update *fqpath* when *file* is updated, using standard view maintenance logic. BOOM-FS defines several other views to compute derived file system metadata, such as the total size of each file and the contents of each directory. The materialization of each view can easily be turned on or off via simple Overlog table definition statements. During the development process, we regularly adjusted view materialization to trade off read performance against write performance and storage requirements.

At each DataNode, chunks are stored as regular files on the file system. In addition, each DataNode maintains a relation describing the chunks stored at that node. This relation is populated by periodically invoking a table function defined in Java that walks the appropriate directory of the DataNode's local file system.

[2]The order of a file's chunks must also be specified, because relations are unordered. Currently, we assign chunk IDs in a monotonically increasing fashion and only support append operations, so clients can determine a file's chunk order by sorting chunk IDs.

```
// The set of nodes holding each chunk
compute_chunk_locs(ChunkId, set<NodeAddr>) :-
    hb_chunk(NodeAddr, ChunkId, _);

// Chunk exists => return success and set of nodes
response(@Src, RequestId, true, NodeSet) :-
    request(@Master, RequestId, Src,
            "ChunkLocations", ChunkId),
    compute_chunk_locs(ChunkId, NodeSet);

// Chunk does not exist => return failure
response(@Src, RequestId, false, null) :-
    request(@Master, RequestId, Src,
            "ChunkLocations", ChunkId),
    notin hb_chunk(_, ChunkId, _);
```

Figure 2: NameNode rules to return the set of DataNodes that hold a given chunk in BOOM-FS.

System	*Lines of Java*	*Lines of Overlog*
HDFS	~21,700	0
BOOM-FS	1,431	469

Table 4: Code size of two file system implementations.

3.2.3 Communication Protocols

BOOM-FS uses three different protocols: the *metadata protocol* which clients and NameNodes use to exchange file metadata, the *heartbeat protocol* which DataNodes use to notify the NameNode about chunk locations and DataNode liveness, and the *data protocol* which clients and DataNodes use to exchange chunks. We implemented the metadata and heartbeat protocols with a set of distributed Overlog rules in a similar style. The data protocol was implemented in Java because it is simple and performance critical. We proceed to describe the three protocols in order.

For each command in the metadata protocol, there is a single rule at the client (stating that a new request tuple should be "stored" at the NameNode). There are typically two corresponding rules at the NameNode: one to specify the result tuple that should be stored at the client, and another to handle errors by returning a failure message. An example of the NameNode rules for Chunk Location requests is shown in Figure 2.

Requests that modify metadata follow the same basic structure, except that in addition to deducing a new result tuple at the client, the NameNode rules also deduce changes to the file system metadata relations. Concurrent requests are serialized by JOL at the NameNode. While this simple approach has been sufficient for our experiments, we plan to explore more sophisticated concurrency control techniques in the future.

DataNode heartbeats have a similar request/response pattern, but are not driven by the arrival of network events. Instead, they are "clocked" by joining with the built-in *periodic* relation [18], which produces new tuples at every tick of a wall-clock timer. In addition, control protocol messages from the NameNode to DataNodes are deduced when certain system invariants are unmet; for example, when the number of replicas for a chunk drops below the configured replication factor.

Finally, the data protocol is a straightforward mechanism for transferring the content of a chunk between clients and DataNodes. This protocol is orchestrated by Overlog rules but implemented in Java. When an Overlog rule deduces that a chunk must be transferred from host *X* to *Y*, an output event is triggered at *X*. A Java event handler at *X* listens for these output events, and uses a simple but efficient data transfer protocol to send the chunk to host *Y*. To implement this protocol, we wrote a simple multi-threaded server in Java that runs on the DataNodes.

3.2.4 Discussion

After two months of work, we had a working implementation of metadata handling strictly in Overlog, and it was straightforward to add Java code to store chunks in UNIX files. Adding the necessary Hadoop client APIs in Java took an additional week. Adding metadata durability took about a day. Table 4 compares statistics about the code bases of BOOM-FS and HDFS. The DataNode implementation accounts for 414 lines of the Java in BOOM-FS; the remainder is mostly devoted to system configuration, bootstrapping, and a client library. Adding support for accessing BOOM-FS to Hadoop required an additional 400 lines of Java.

BOOM-FS has performance, scaling and failure-handling properties similar to those of HDFS; again, we provide a simple performance validation in Appendix B. By following the HDFS architecture, BOOM-FS can tolerate DataNode failures but has a single point of failure and scalability bottleneck at the NameNode.

Like MapReduce, HDFS is actually a fairly simple system, and we feel that BOOM-FS reflects that simplicity well. HDFS sidesteps many of the performance challenges of traditional file systems and databases by focusing nearly exclusively on scanning large files. It avoids most distributed systems challenges regarding replication and fault-tolerance by implementing coordination with a single centralized NameNode. As a result, most of our implementation consists of simple message handling and management of the hierarchical file system namespace. Datalog materialized view logic was not hard to implement in JOL, and took care of most of the performance issues we faced over the course of our development.

4. THE AVAILABILITY REV

Having achieved a fairly faithful implementation of MapReduce and HDFS, we were ready to explore one of our main motivating hypotheses: that data-centric programming would make it easy to add complex distributed functionality to an existing system. We chose an ambitious goal: retrofitting BOOM-FS with high availability failover via "hot standby" NameNodes. A proposal for warm standby was posted to the Hadoop issue tracker in October of 2008 ([17] issue HADOOP-4539). We felt that a hot standby scheme would be more useful, and we deliberately wanted to pick a more challenging design to see how hard it would be to build in Overlog.

4.1 Paxos Implementation

Correctly implementing efficient hot standby replication is tricky, since replica state must remain consistent in the face of node failures and lost messages. One solution to this problem is to implement a globally-consistent distributed log, which guarantees a total ordering over events affecting replicated state. The Paxos algorithm is the canonical mechanism for this feature [16].

We began by creating an Overlog implementation of basic Paxos, focusing on correctness and adhering as closely as possible to the initial specification. Our first effort resulted in an impressively short program: 22 Overlog rules in 53 lines of code. We found that Overlog was a good fit for this task: our Overlog rules corresponded nearly line-for-line with the statements of invariants from Lamport's original paper [16]. Our entire implementation fit on a single screen, so its faithfulness to the original specification could be visually confirmed. To this point, working with a data-centric language was extremely gratifying.

We then needed to convert basic Paxos into a working primitive for a distributed log. This required adding the ability to pass a series of log entries ("Multi-Paxos"), a liveness module, and a catchup algorithm, as well as optimizations to reduce message complexity. This caused our implementation to swell to 50 rules in roughly 400

lines of code. As noted in the Google implementation [6], these enhancements made the code considerably more difficult to check for correctness. Our code also lost some of its pristine declarative character. This was due in part to the evolution of the Paxos research papers: while the original Paxos was described as a set of invariants over state, most of the optimizations were described as transition rules in state machines. Hence we found ourselves translating state-machine pseudocode back into logical invariants, and it took some time to gain confidence in our code. The resulting implementation is still very concise relative to a traditional programming language, but it highlighted the difficulty of using a data-centric programming model for complex tasks that were not originally specified that way. We return to this point in Section 7.

4.2 BOOM-FS Integration

Once we had Paxos in place, it was straightforward to support the replication of the distributed file system metadata. All state-altering actions are represented in the revised BOOM-FS as Paxos decrees, which are passed into the Paxos logic via a single Overlog rule that intercepts tentative actions and places them into a table that is joined with Paxos rules. Each action is considered complete at a given site when it is "read back" from the Paxos log, i.e. when it becomes visible in a join with a table representing the local copy of that log. A sequence number field in the Paxos log table captures the globally-accepted order of actions on all replicas.

We also had to ensure that Paxos state was durable in the face of crashes. The support for persistent tables in Overlog made this straightforward: Lamport's description of Paxos explicitly distinguishes between transient and durable state. Our implementation already divided this state into separate relations, so we simply marked the appropriate relations as durable.

We validated the performance of our implementation experimentally: in the absence of failure replication has negligible performance impact, but when the primary NameNode fails, a backup NameNode takes over reasonably quickly. This is discussed in more detail in Appendix D.

4.3 Discussion

The Availability revision was our first foray into serious distributed systems programming, and we continued to benefit from the high-level abstractions provided by Overlog. Most of our attention was focused at the appropriate level of complexity: faithfully capturing the reasoning involved in distributed protocols.

Lamport's original paper describes Paxos as a set of logical invariants, which he uses in his proof of correctness. Translating these into Overlog rules was a straightforward exercise in declarative programming. Each rule covers a potentially large portion of the state space, drastically simplifying the case-by-case transitions that would have to be specified in a state machine-based implementation. However, choosing an invariant-based approach made it harder to adopt optimizations from the literature, because these were often specified as state machines. For example, a common optimization of basic Paxos avoids the high messaging cost of reaching quorum by skipping the protocol's first phase once a master has established quorum: subsequent decrees then use the established quorum, and merely hold rounds of voting while steady state is maintained. This is naturally expressed in a state machine model as a pair of transition rules for the same input (a request) given different starting states. In our implementation, we frequently found it easier in such cases to model the state as a relation with a single row, allow certain rules to fire only in certain states, and explicitly describe the transitions, rather than to reformulate the optimizations in terms of original logic. Though mimicking a state machine is straightforward, the resulting rules have a hybrid feel, which somewhat compromises our high-level protocol specification.

5. THE SCALABILITY REV

HDFS NameNodes manage large amounts of file system metadata, which is kept in memory to ensure good performance. The original GFS paper acknowledged that this could cause significant memory pressure [12], and NameNode scaling is often an issue in practice at Yahoo!. Given the data-centric nature of BOOM-FS, we hoped to very simply scale out the NameNode across multiple *NameNode-partitions*. From a database design perspective this seemed trivial — it involved adding a "partition" column to some Overlog tables. The resulting code composes cleanly with our availability implementation: each NameNode-partition can be a single node, or a Paxos group.

There are many options for partitioning the files in a directory tree. We opted for a simple strategy based on the hash of the fully qualified pathname of each file. We also modified the client library to broadcast requests for directory listings and directory creation to each NameNode-partition. Although the resulting directory creation implementation is not atomic, it is idempotent; recreating a partially-created directory will restore the system to a consistent state, and will preserve any files in the partially-created directory.

For all other BOOM-FS operations, clients have enough information to determine the correct NameNode-partition. We do not support atomic "move" or "rename" across partitions. This feature is not exercised by Hadoop, and complicates distributed file system implementations considerably. In our case, it would involve the atomic transfer of state between otherwise-independent Paxos instances. We believe this would be relatively clean to implement — we have a two-phase commit protocol implemented in Overlog — but decided not to pursue this feature at present.

5.1 Discussion

By isolating the file system state into relations, it became a textbook exercise to partition that state across nodes. It took 8 hours of developer time to implement NameNode partitioning; two of these hours were spent adding partitioning and broadcast support to the Overlog code. This was a clear win for the data-centric approach.

The simplicity of file system scale-out made it easy to think through its integration with Paxos, a combination that might otherwise seem very complex. Our confidence in being able to compose techniques from the literature is a function of the compactness and resulting clarity of our code.

6. THE MONITORING REV

As our BOOM Analytics prototype matured and we began to refine it, we started to suffer from a lack of performance monitoring and debugging tools. Singh et al. pointed out that Overlog is well-suited to writing distributed monitoring queries, and offers a naturally introspective approach: simple Overlog queries can monitor complex protocols [25]. Following that idea, we decided to develop a suite of debugging and monitoring tools for our own use.

6.1 Invariants

One advantage of a logic-oriented language like Overlog is that it encourages the specification of system invariants, including "watchdogs" that provide runtime checks of behavior induced by the program. For example, one can confirm that the number of messages sent by a protocol like Paxos matches the specification. Distributed Overlog rules induce asynchrony across nodes; such rules are only *attempts* to achieve invariants. An Overlog program needs to be en-

hanced with global coordination mechanisms like two-phase commit or distributed snapshots to convert distributed Overlog rules into global invariants [7]. Singh et al. have shown how to implement Chandy-Lamport distributed snapshots in Overlog [25]; we did not go that far in our own implementation.

To simplify debugging, we wanted a mechanism to integrate Overlog invariant checks into Java exception handling. To this end, we added a relation called *die* to JOL; when tuples are inserted into the *die* relation, a Java event listener is triggered that throws an exception. This feature makes it easy to link invariant assertions in Overlog to Java exceptions: one writes an Overlog rule with an invariant check in the body, and the *die* relation in the head.

We made extensive use of these local-node invariants in our code and unit tests. Although these invariant rules increase the size of a program, they improve readability in addition to reliability. This is important in a language like Overlog: it is a terse language, and program complexity grows rapidly with code size. Assertions that we specified early in the implementation of Paxos aided our confidence in its correctness as we added features and optimizations.

6.2 Monitoring via Metaprogramming

Our initial prototypes of both BOOM-MR and BOOM-FS had significant performance problems. Unfortunately, Java-level performance tools were of little help. A poorly-tuned Overlog program spends most of its time in the same routines as a well-tuned Overlog program: in dataflow operators like Join and Aggregation. Java-level profiling lacks the semantics to determine which rules are causing the lion's share of the dataflow code invocations.

Fortunately, it is easy to do this kind of bookkeeping directly in Overlog. In the simplest approach, one can replicate the body of each rule in an Overlog program and send its output to a log table (which can be either local or remote). For example, the Paxos rule that tests whether a particular round of voting has reached quorum:

```
quorum(@Master, Round) :-
    priestCnt(@Master, Pcnt),
    lastPromiseCnt(@Master, Round, Vcnt),
    Vcnt > (Pcnt / 2);
```

might have an associated tracing rule:

```
trace_r1(@Master, Round, RuleHead, Tstamp) :-
    priestCnt(@Master, Pcnt),
    lastPromiseCnt(@Master, Round, Vcnt),
    Vcnt > (Pcnt / 2),
    RuleHead = "quorum",
    Tstamp = System.currentTimeMillis();
```

This approach captures per-rule dataflow in a trace relation that can be queried later. Finer levels of detail can be achieved by "tapping" each of the predicates in the rule body separately in a similar fashion. The resulting program passes no more than twice as much data through the system, with one copy of the data being "teed off" for tracing along the way. When profiling, this overhead is often acceptable. However, writing the trace rules by hand is tedious.

Using the metaprogramming approach of Evita Raced [8], we were able to automate this task via a *trace rewriting* program written in Overlog, involving the meta-tables of rules and terms. The trace rewriting expresses logically that for selected rules of some program, new rules should be added to the program containing the body terms of the original rule, and auto-generated head terms. Network traces fall out of this approach naturally: any dataflow transition that results in network communication is flagged in the generated head predicate during trace rewriting.

Using this idea, it took less than a day to create a general-purpose Overlog code coverage tool that traced the execution of our unit tests and reported statistics on the "firings" of rules in the JOL runtime, and the counts of tuples deduced into tables. Our metaprogram for code coverage and network tracing consists of 5 Overlog rules that are evaluated by every participating node, and 12 summary rules that are run at a centralized location. Several hundred lines of Java implement a rudimentary front end to the tool. We ran our regression tests through this tool, and immediately found both "dead code" rules in our programs, and code that we knew needed to be exercised by the tests but was as-yet uncovered.

6.3 Logging

Hadoop comes with fairly extensive logging facilities that can track not only logic internal to the application, but performance counters that capture the current state of the worker nodes.

TaskTrackers write their application logs to a local disk and rely on an external mechanism to collect, ship and process these logs; Chukwa is one such tool used in the Hadoop community [3]. In Chukwa, a local *agent* written in Java implements a number of *adaptors* that gather files (e.g., the Hadoop log) and the output of system utilities (e.g. top, iostat), and forward the data to intermediaries called *collectors*, which in turn buffer messages before forwarding them to *data sinks*. At the data sinks, the unstructured log data is eventually parsed by a MapReduce job, effectively redistributing it over the cluster in HDFS.

We wanted to prototype similar logging facilities in Overlog, not only because it seemed an easy extension of the existing infrastructure, but because it would close a feedback loop that — in future — could allow us to make more intelligent scheduling and placement decisions. Further, we observed that the mechanisms for forwarding, buffering, aggregation and analysis of streams are already available via Overlog.

We began by implementing Java modules that read from the `/proc` file system and produce the results as JOL tuples. We also wrote Java modules to convert Hadoop application logs into tuples. Windowing, aggregation and buffering are carried out in Overlog, as are the summary queries run at the data sinks.

In-network buffering and aggregation were simple to implement in Overlog, and this avoided the need to add explicit intermediary processes to play the role of collectors. The result was a very simple implementation of the general Chukwa idea. We implemented the "agent" and "collector" logic via a small set of rules that run inside the same JOL runtime as the NameNode process. This made our logger easy to write, well-integrated into the rest of the system, and easily extensible. On the other hand, it puts the logging mechanism on the runtime's critical path, and is unlikely to scale as well as Chukwa as log sizes increase. For our purposes, we were primarily interested in gathering and acting quickly upon telemetry data, and the current collection rates are reasonable for the existing JOL implementation. We are investigating alternative data forwarding pathways like those we used for BOOM-FS for the bulk forwarding of application logs, which are significantly larger and are not amenable to in-network aggregation.

7. EXPERIENCE AND LESSONS

Our overall experience with BOOM Analytics has been very positive. Building the system required only nine months of part-time work by four developers. We have been frankly surprised at our own productivity, and even with a healthy self-regard we cannot attribute it to our programming skills per se. Along the way, there have been some interesting lessons learned, and a bit of time for initial reflections on the process.

7.1 Everything Is Data

The most positive aspects of our experience with Overlog and

BOOM Analytics came directly from data-centric programming. In the system we built, *everything* is data, represented as tuples in tables. This includes traditional persistent information like file system metadata, runtime state like TaskTracker status, summary statistics like those used by the JobTracker's scheduling policy, in-flight messages, system events, execution state of the system, and even parsed code.

The benefits of this approach are perhaps best illustrated by the extreme simplicity with which we scaled out the NameNode via partitioning (Section 5): by having the relevant state stored as data, we were able to use standard data partitioning to achieve what would ordinarily be a significant rearchitecting of the system. Similarly, the ease with which we implemented system monitoring — via both system introspection tables and rule rewriting — arose because we could easily write rules that manipulated concepts as diverse as transient system state and program semantics (Section 6).

The uniformity of data-centric interfaces also enables *interposition* [14] of components in a natural manner: the dataflow "pipe" between two system modules can be easily rerouted to go through a third module. This enabled the simplicity of incorporating our Overlog LATE scheduler into BOOM-MR (Section 3.1.2). Because dataflows can be routed across the network (via the location specifier in a rule's head), interposition can also involve distributed logic — this is how we easily added Paxos support into the BOOM-FS NameNode (Section 4). Our experience suggests that a form of encapsulation could be achieved by constraining the points in the dataflow at which interposition is allowed to occur.

The last data-centric programming benefit we observed related to the timestepped dataflow execution model, which we found to be simpler than traditional notions of concurrent programming. Traditional models for concurrency include event loops and multi-threaded programming. Our concern regarding event loops — and the state machine programming models that often accompany them — is that one needs to reason about *combinations* of states and events. That would seem to put a quadratic reasoning task on the programmer. In principle our logic programming deals with the same issue, but we found that each composition of two tables (or tuple-streams) could be thought through in isolation, much as one thinks about composing relational operators or piping Map and Reduce tasks. Given our prior experience writing multi-threaded code with locking, we were happy that the simple timestep model of Overlog obviated the need for this entirely — there is no explicit synchronization logic in any of the BOOM Analytics code, and we view this as a clear victory for the programming model.

In all, none of this discussion seems specific to logic programming per se. We suspect that a more algebraic style of programming — for instance a combination of MapReduce and joins — would afford many of the same benefits as Overlog, if it were pushed to a similar degree of generality.

7.2 Developing in Overlog

We have had various frustrations with the Overlog language: many minor, and a few major. The minor complaints are not technically significant, but one at least seems notable. Some team members grew to dislike Datalog's specification of equijoins (unification) via repetition of variables; it is hard to write, and especially hard to read. A text editor with syntax highlighting helps to some extent, but we suspect that no programming language will grow popular with this syntactic convention. That said, the issue is eminently fixable: SQL's named-field approach is one option, and we can imagine others. In the end, any irritability with Datalog syntax was far outweighed by our positive experience with the productivity offered by Overlog.

Another programming challenge we wrestled with was the translation of state machine programming into logic (Section 4). In fairness, the porting task was not actually very hard: in most cases it amounted to writing message-handling rules in Overlog that had a familiar structure. But upon deeper reflection, our port was shallow and syntactic; the resulting Overlog does not "feel" like logic, in the invariant style of Lamport's original Paxos specification. Having gotten the code working, we hope to revisit it with an eye toward rethinking the global *intent* of the state-machine optimizations. This would not only fit the spirit of Overlog better, but perhaps contribute to a deeper understanding of the ideas involved.

With respect to consistency of storage, we were comfortable with our model of associating a local storage transaction with each fixpoint. However, we expect that this may change as we evolve the use of JOL. For example, we have not to date seriously dealt with the idea of a single JOL runtime hosting multiple programs. We expect this to be a natural desire in our future work.

7.3 Performance

JOL performance was good enough for BOOM Analytics to match Hadoop performance, but we are conscious that it has room to improve. We observed that system load averages were much lower with Hadoop than with BOOM Analytics. We are now exploring a reimplementation of the dataflow kernel of JOL in C, with the goal of having it run as fast as the OS network handling that feeds it. This is not important for BOOM Analytics, but will be important as we consider more interactive cloud infrastructure.

In the interim, we actually think the modest performance of the current JOL interpreter guided us to reasonably good design choices. By using Java for the data path in BOOM-FS, for example, we ended up spending very little of our development time on efficient data transfer. In retrospect, we were grateful to have used that time for more challenging efforts like implementing Paxos.

8. CONCLUSION

We built BOOM Analytics to evaluate three key questions about data-centric programming of clusters: (1) can it radically simplify the prototyping of distributed systems, (2) can it be used to write scalable, performant code, and (3) can it enable a new generation of programmers to innovate on novel cloud computing platforms. Our experience suggests that the answer to the first of these questions is certainly true, and the second is within reach. The third question is unresolved. Overlog in its current form is not going to attract programmers to distributed computing, but we think that its benefits point the way to more pleasant languages that could realistically commoditize distributed programming in the Cloud.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 0722077 and 0713661, the University of California MICRO program, and gifts from Sun Microsystems, Inc. and Microsoft Corporation.

9. REFERENCES

[1] A. Abouzeid et al. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *VLDB*, 2009.

[2] M. P. Ashley-Rollman et al. Declarative Programming for Modular Robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications*, 2007.

[3] J. Boulon et al. Chukwa, a large-scale monitoring system. In *Cloud Computing and its Applications*, pages 1–5, October 2008.

[4] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4), 2001.

[5] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[8] T. Condie et al. Evita Raced: metacompilation for declarative networks. In *VLDB*, 2008.

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

[10] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[11] J. Eisner, E. Goldlust, and N. A. Smith. Dyna: a declarative language for implementing dynamic programs. In *Proc. ACL*, 2004.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[13] H. S. Gunawi et al. SQCK: A Declarative File System Checker. In *OSDI*, 2008.

[14] M. B. Jones. Interposition agents: transparently interposing user code at the system interface. In *SOSP*, 1993.

[15] M. S. Lam et al. Context-sensitive program analysis as database queries. In *PODS*, 2005.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[17] Hadoop jira issue tracker, July 2009. `http://issues.apache.org/jira/browse/HADOOP`.

[18] B. T. Loo et al. Implementing declarative overlays. In *SOSP*, 2005.

[19] B. T. Loo et al. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[20] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[21] W. R. Marczak et al. Declarative reconfigurable trust management. In *CIDR*, 2009.

[22] Nokia Corporation. disco: massive data – minimal code, 2009. `http://discoproject.org/`.

[23] T. Schutt et al. Scalaris: Reliable transactional P2P key/value store. In *SIGPLAN Workshop on Erlang*, 2008.

[24] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *OSDI*, 2006.

[25] A. Singh et al. Using queries for distributed monitoring and forensics. In *EuroSys*, 2006.

[26] M. Stonebraker. Inclusion of new types in relational data base systems. In *ICDE*, 1986.

[27] The Hive Project. Hive home page, 2009. `http://hadoop.apache.org/hive/`.

[28] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[29] W. White et al. Scaling games to epic proportions. In *SIGMOD*, 2007.

[30] F. Yang et al. Hilda: A high-level language for data-driven web applications. In *ICDE*, 2006.

[31] Y. Yu et al. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[32] M. Zaharia et al. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

APPENDIX

A. DATALOG BACKGROUND

The Datalog language is defined over relational tables; it is a purely logical query language that makes no changes to the stored tables. A Datalog *program* is a set of *rules* or named queries, in the spirit of SQL's *view*s. A simple Datalog rule has the form:

$$r_{head}(\langle col\text{-}list \rangle) \texttt{ :- } r_1(\langle col\text{-}list \rangle), \ldots, r_n(\langle col\text{-}list \rangle)$$

Each term r_i represents a relation, either stored (a database table) or derived (the result of other rules). Relations' columns are listed as a comma-separated list of variable names; by convention, variables begin with capital letters. Terms to the right of the `:-` symbol form the rule *body* (corresponding to the `FROM` and `WHERE` clauses in SQL), the relation to the left is called the *head* (corresponding to the `SELECT` clause in SQL). Each rule is a logical assertion that the head relation contains those tuples that can be generated from the body relations. Tables in the body are *unified* (joined together) based on the positions of the repeated variables in the column lists of the body terms. For example, a canonical Datalog program for recursively computing paths from links [19] is shown in Figure 3 (ignoring the Overlog-specific `@` notation), along with analogous SQL for the inductive rule. Note how the SQL `WHERE` clause corresponds to the repeated use of the variable `To` in the Datalog.

```
path(@From, To, To, Cost)
        :- link(@From, To, Cost);
path(@From, End, To, Cost1+Cost2)
        :- link(@From, To, Cost1),
           path(@To, End, NextHop, Cost2);

WITH path(Start, End, NextHop, Cost) AS
  ( SELECT link.From, path.End,
           link.To, link.Cost+path.Cost
      FROM link, path
     WHERE link.To = path.Start );
```

Figure 3: Example Overlog for computing paths from links, along with an SQL translation of the second rule.

B. VALIDATION OF INITIAL PROTOTYPE

While improved performance was not a goal of our work, we wanted to ensure that the performance of BOOM Analytics was competitive with Hadoop. To that end, we conducted a series of performance experiments using a 101-node cluster on Amazon EC2. One node executed the Hadoop JobTracker and the DFS NameNode, while the remaining 100 nodes served as slaves for running the Hadoop TaskTrackers and DFS DataNodes. The master node ran on an "high-CPU extra large" EC2 instance with 7.2 GB of memory and 8 virtual cores. Our slave nodes executed on "high-CPU medium" EC2 instances with 1.7 GB of memory and 2 virtual cores. Each virtual core is the equivalent of a 2007-era 2.5Ghz Intel Xeon processor.

For our experiments, we compared BOOM Analytics with Hadoop 18.1. Our workload was a wordcount job on a 30 GB file. The wordcount job consisted of 481 map tasks and 100 reduce tasks. Each of the 100 slave nodes hosted a single TaskTracker instance that can support the simultaneous execution of 2 map tasks and 2 reduce tasks.

Figure 5 contains four performance graphs, comparing the performance of different combinations of Hadoop MapReduce, HDFS, BOOM-MR, and BOOM-FS. Each graph reports a cumulative distribution of map and reduce task completion times (in seconds). The map tasks complete in three distinct "waves". This is because

```
// fqpath: Fully-qualified paths.
// Base case: root directory has null parent
fqpath(Path, FileId) :-
    file(FileId, FParentId, _, true),
    FParentId = null, Path = "/";

fqpath(Path, FileId) :-
    file(FileId, FParentId, FName, _),
    fqpath(ParentPath, FParentId),
    // Do not add extra slash if parent is root dir
    PathSep = (ParentPath = "/" ? "" : "/"),
    Path = ParentPath + PathSep + FName;
```

Figure 4: Example Overlog for computing fully-qualified pathnames from the base file system metadata in BOOM-FS.

only 2×100 map tasks can be scheduled at once. Although all 100 reduce tasks can be scheduled immediately, no reduce task can finish until all maps have been completed, because each reduce task requires the output of all map tasks.

The upper-left graph describes the performance of Hadoop running on top of HDFS, and hence serves as a baseline for the subsequent graphs. The lower-left graph details BOOM-MR running over HDFS. This graph shows that map and reduce task completion times under BOOM-MR are nearly identical to Hadoop 18.1. The upper-right and lower-right graphs detail the performance of Hadoop MapReduce and BOOM-MR running on top of BOOM-FS, respectively. BOOM-FS performance is slightly worse than HDFS, but remains very competitive.

C. MAPREDUCE SCHEDULING

MapReduce scheduling has been the subject of recent research, and one of our early motivations for building BOOM Analytics was to make that research extremely easy to carry out. In our initial BOOM-MR prototype, we implemented Hadoop's default First-Come-First-Served policy for task scheduling, which was captured in 9 rules (96 lines) of scheduler policy. Next, we implemented the recently-proposed LATE policy [32] to evaluate both (a) the difficulty of prototyping a new policy, and (b) the faithfulness of our Overlog-based execution to that of Hadoop using two separate scheduling algorithms.

The LATE policy presents an alternative scheme for speculative task execution on *straggler* tasks [32], in an effort to improve on Hadoop's policy. There are two aspects to each policy: choosing which tasks to speculatively re-execute, and choosing TaskTrackers to run those tasks. Original Hadoop re-executes a task if its progress is more than 0.2 (on a scale of $[0..1]$) below the mean progress of similar tasks; it assigns speculative tasks using the same policy as it uses for initial tasks. LATE chooses tasks to re-execute via an *estimated finish time* metric based on the task's *progress rate*. Moreover, it avoids assigning speculative tasks to TaskTrackers that exhibit slow performance executing similar tasks, in hopes of preventing the creation of new stragglers.

The LATE policy is specified in the paper via just three lines of pseudocode, which make use of three performance statistics called *SlowNodeThreshold*, *SlowTaskThreshold*, and *SpeculativeCap*. The first two of these statistics correspond to the 25th percentiles of progress rates across TaskTrackers and across tasks, respectively. The *SpeculativeCap* is suggested to be set at 10% of available task slots [32]. We compute these thresholds via the five Overlog rules shown in Figure 6. Integrating the rules into BOOM-MR required modifying two additional Overlog rules that identify tasks to speculatively re-execute, and that choose TaskTrackers for scheduling those tasks.

Figure 7 shows the cumulative distribution of the completion

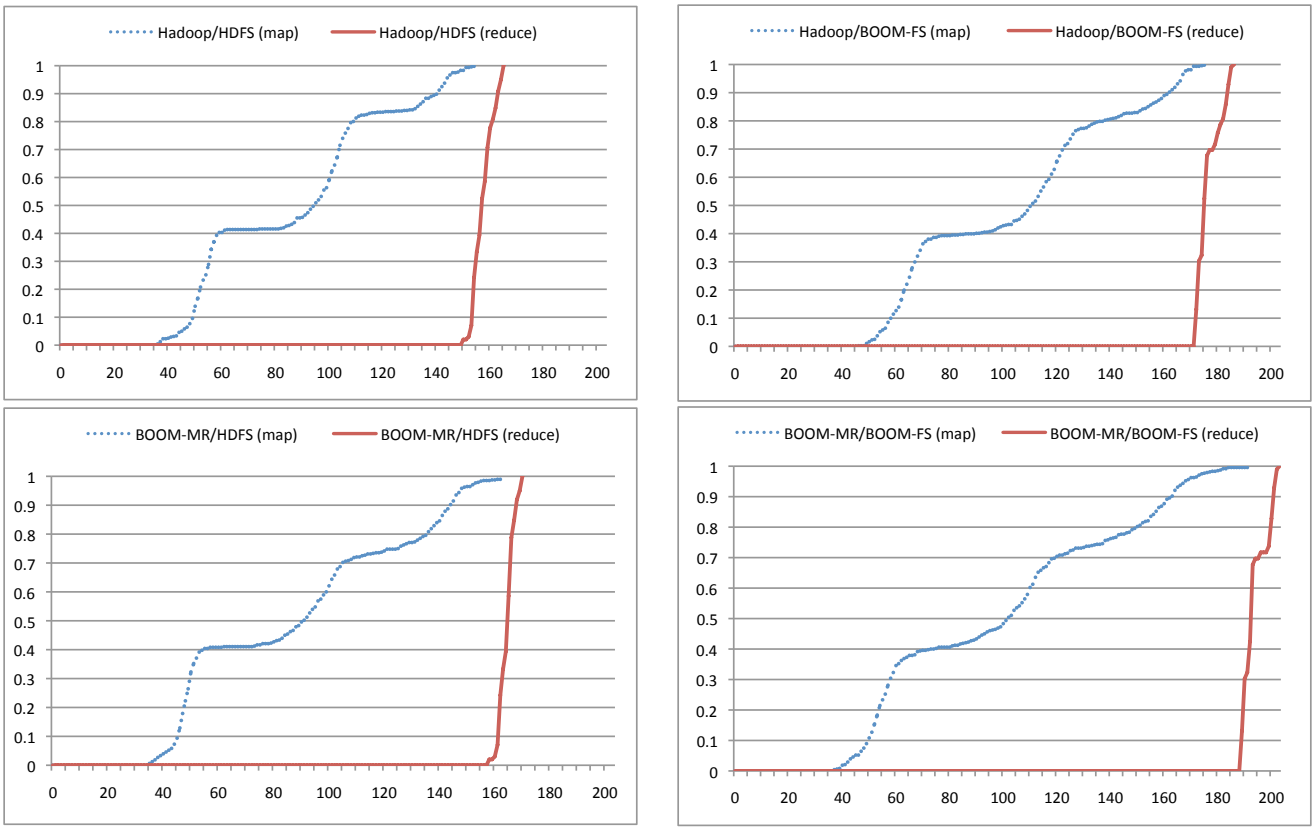

Figure 5: CDF of map and reduce task completion for Hadoop and BOOM-MR over HDFS and BOOM-FS. In all graphs, the horizontal axis is elapsed time in seconds, and the vertical represents % of tasks completed.

time for reduce task executions on EC2 under normal load, and with artificial extra load placed on six straggler nodes. The same wordcount workload was used for this experiment but the number of reduce tasks was increased from 100 to 400 in order to produce two waves of reduce tasks. The plots labeled "No Stragglers" represent normal load. The plots labeled "Stragglers" and "Stragglers (LATE)" are taken under the (six node) artificial load using the vanilla Hadoop and LATE policies (respectively) to identify speculative tasks. We do not show a CDF of the map task execution time since the artificial load barely affects it — the six stragglers have no effect on other map tasks, they just result in a slower growth from just below 100% to completion. The first wave of 200 reduce tasks is scheduled concurrently with all the map tasks. This first wave of reduce tasks will not finish until all map tasks have completed, which increases the completion time of these tasks as indicated in the right portion of the graph. The second wave of 200 reduce tasks will not experience the delay due to unfinished map work since it is scheduled after all map tasks have finished. These shorter completion times are reported in the left portion of the graph. Furthermore, stragglers have less of an impact on the second wave of reduce tasks since less work (i.e., no map work) is being performed. Figure 7 shows this effect, and also demonstrates how the LATE implementation in BOOM Analytics handles stragglers much more effectively than the default speculation policy ported from Hadoop. This echoes the results of Zaharia et al. [32]

D. VALIDATION: HIGH AVAILABILITY

After adding support for high availability to the BOOM-FS NameNode (Section 4), we wanted to evaluate two properties of our implementation. At a fine-grained level, we wanted to ensure that our complete Paxos implementation was operating according to the specifications in the literature. This required logging and analyzing network messages sent during the Paxos protocol. This was a natural fit for the metaprogrammed tracing tools we discussed in Section 6.2. We created unit tests to trace the message complexity of our Paxos code, both at steady state and under churn. When the message complexity of our implementation matched the specification, we had more confidence in the correctness of our code.

Second, we wanted to see the availability feature "in action", and to get a sense of how our implementation would perform in the face of master failures. Specifically, we evaluated the impact of the consensus protocol on BOOM Analytics system performance, and the effect of failures on overall completion time. We ran a Hadoop wordcount job on a 5GB input file with a cluster of 20 EC2 nodes, varying the number of master nodes and the failure condition. These results are summarized in Table 5. We then used the same workload to perform a set of simple fault-injection experiments to measure the effect of primary master failures on job completion rates at a finer grain, observing the progress of the map and reduce jobs involved in the wordcount program. Figure 8 shows the cumulative distribution of the percentage of completed map and reduce jobs over time, in normal operation and with a failure of the primary NameNode during the map phase. Note that Map tasks read from HDFS and write to local files, whereas Reduce tasks read from Mappers and write to HDFS. This explains why the CDF for Reduce tasks under failure goes so crisply flat in Figure 8: while failover is underway after an HDFS NameNode failure, some nearly-finished Map tasks may be able to complete, but no Reduce task can complete.

```
// Compute progress rate per task
taskPR(JobId, TaskId, Type, ProgressRate) :-
    task(JobId, TaskId, Type, _, _, _, Status),
    Status.state() != FAILED,
    Time = Status.finish() > 0 ?
        Status.finish() : currentTimeMillis(),
    ProgressRate = Status.progress() /
                   (Time - Status.start());

// For each job, compute 25th pctile rate across tasks
taskPRList(JobId, Type, percentile<0.25, PRate>) :-
    taskPR(JobId, TaskId, Type, PRate);

// Compute progress rate per tracker
trackerPR(Tracker, JobId, Type, avg<PRate>) :-
    task(JobId, TaskId, Type, _),
    taskAttempt(JobId, TaskId, _, Progress, State,
                Phase, Tracker, Start, Finish),
    State != FAILED,
    Time = Finish > 0 ? Finish : currentTimeMillis(),
    PRate = Progress / (Time - Start);

// For each job, compute 25th pctile rate across trackers
trackerPRList(JobId, Type, percentile<0.25, AvgPRate>) :-
    trackerPR(_, JobId, Type, AvgPRate);

// Compute available map/reduce slots
speculativeCap(sum<MapSlots>, sum<ReduceSlots>) :-
    taskTracker(_, _, _, _, _, _,
                MapCount, ReduceCount,
                MaxMap, MaxReduce),
    MapSlots = MaxMap - MapCount,
    ReduceSlots = MaxReduce - ReduceCount;
```

Figure 6: Overlog to compute statistics for LATE.

Number of NameNodes	Failure Condition	Avg. Completion Time (secs)	Standard Deviation
1	None	101.89	12.12
3	None	102.70	9.53
3	Backup	100.10	9.94
3	Primary	148.47	13.94

Table 5: Job completion times with a single NameNode, 3 Paxos-enabled NameNodes, backup NameNode failure, and primary NameNode failure.

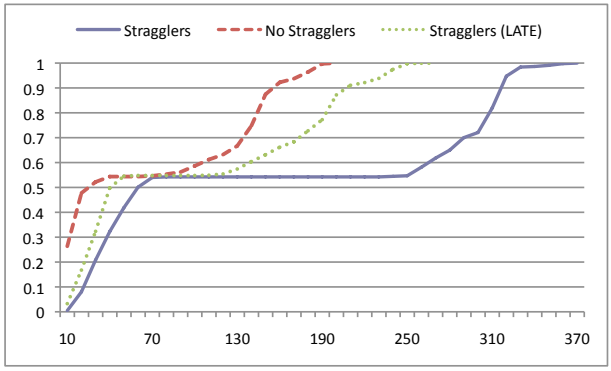

Figure 7: CDF of reduce task completion times (secs), with and without stragglers.

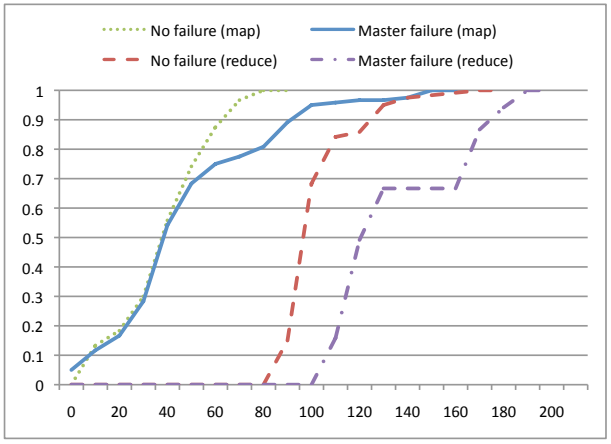

Figure 8: CDF of completed tasks over time (secs), with and without primary master failure.