BOOM: Data-Centric Programming in the Datacenter

Draft. Please do not distribute

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

The advent of cloud computing is turning datacenter clusters into a commodity. By making large clusters trivial to acquire, manage, and maintain, cloud computing promises to seed a phase of innovative software development, with a wide range of programmers developing new services that scale out quickly and flexibly. However, current cloud platforms focus on relatively traditional, single-server programming models over shared storage, and do little to simplify the task of coordinating large-scale distributed systems. There is as yet no widely-used programming model that lets a developer easily harness the distributed power of a cluster.

In this paper, we detail our experience building distributed datacenter software via high-level, data-centric programming. Using the Overlog language and Java, we developed a "Big Data" analytics stack that is API-compatible with Hadoop and HDFS. We describe our experience reimplementing the Hadoop stack and extending it incrementally with new features not yet available in Hadoop, including availability, scalability, and unique monitoring and debugging facilities. Developed in a relatively short nine-month design cycle, our Overlog interpreter and Hadoop implementation perform within a modest factor of the standard Java-only implementation, with a compact and easily-extendible codebase. We reflect on the opportunities and challenges we encountered along the way, which may inform new development environments for distributed programming in datacenters.

1. INTRODUCTION

Cluster computing has 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., [13, 16, 2, 30, 31, 8, 14]). This infrastructure consists of distributed software, which is written to manage tricky issues including parallelism, communication, failure, and system resizing. Cluster infrastructure systems support basic service operation, and facilitate software development by other in-house developers.

In recent years, the usage model for clusters has been expanding via *cloud computing* initiatives, which aim to enable third-party developers to host their applications on managed clusters. One goal of fostering a broad developer community is to encourage innovative use of a platform's unique functionality. In the datacenter, a key new feature is the power of multiple coordinated machines. However, the initial cloud platforms have been conservative in the interface they provide to developers: they are "virtualized legacy" development environments, which take traditional single-node programming interfaces into an environment of hosted virtual machines and shared storage. Specifically, Amazon's 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 [13] and Hadoop [31], which has successfully allowed 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 nine months we have been exploring the use of a more general-purpose data-centric approach to datacenter programming. Reviewing some of the initial datacenter infrastructure efforts in the literature (e.g., [13, 16, 8, 14]), 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 operations in memory. We speculated that a high-level data-centric programming language would be well-suited to those tasks, and could significantly ease the development of datacenter software without introducing major computational bottlenecks. We set out to convince ourselves of this idea in a realistic setting.

As starting points, we considered practical lessons from the success of MapReduce and SQL in harnessing parallelism, even though they are not general-purpose programming languages. We also examined the declarative domain-specific languages that have emerged in an increasing variety of research communities in recent years (Section 1.3). Among the proposals in the literature, the most natural for our purposes

seemed to be the Overlog language introduced in the P2 system [27]. Although we were not fond of its formal syntax, it had a number of advantages in our setting: it includes example code for network protocol specification, it has been shown to be useful for distributed coordination protocols [38], and offers an elegant metaprogramming framework for static analysis, program rewriting, and generation of runtime invariant checks [11]. Rather than counting on the initial P2 implementation, we developed our own Java-based Overlog runtime we call *JOL* (Section 2).

To evaluate the use of a data-centric language for the datacenter, we wanted to implement a realistic distributed application. We chose to develop *BOOM*: an API-compliant reimplementation of the Hadoop MapReduce engine and its HDFS distributed file system.[1] In writing BOOM, we preserved the Java API "skin" of Hadoop and HDFS, but replaced their complex internals with Overlog. The Hadoop stack appealed to us for three reasons. First, it exercises the distributed power of a cluster. Unlike a farm of independent web service instances, the Hadoop and HDFS code entail non-trivial coordination among large numbers of nodes, including scheduling, consensus, data replication, and failover. Second, Hadoop appealed as a litmus test, challenging us to build a popular data-centric runtime out of a data-centric language. This kind of challenge is common in the design of new programming languages (e.g. JVMs implemented in Java [1, 41]), and has an echo of more deliberate efforts to co-evolve languages and platforms (C/UNIX, Mesa/Alto, etc). None of the current environments for data-center programming approaches this goal: you cannot easily implement Hadoop via MapReduce, nor Azure or Google App Engine via their interfaces. Finally, Hadoop is a work in progress, still missing significant distributed features like availability and scalability of master nodes. These features served as specific targets for our effort.

1.2 Contributions

The bulk of this paper describes our experience implementing and evolving BOOM, and running it in the EC2 cloud computing environment. After nine months of development, our BOOM prototype runs with modest overheads relative to vanilla Hadoop/HDFS, 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 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 — via limitations of Overlog and JOL — issues that may contribute to an even better programming environment for datacenter development.

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

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 [13, 31] 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 [27], three-tier web services [43], natural language processing [15], modular robotics [3], video games [42], file system metadata analysis [19], and compiler analysis [23].

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 [42] — are *algebraic* (*dataflow*) languages, used to describe the composition and extension of a small dataflow 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. Algebraic dataflow languages are amenable to set-oriented optimization techniques developed for declarative languages [17, 42]. Declarative and dataflow languages can also share the same runtime, as demonstrated by recent integrations of MapReduce and SQL-like syntax in Hive [39] and DryadLINQ [44], and commercial integrations of SQL and MapReduce [18, 35].

Concurrent with our work, the Erlang language was used to implement a simple MapReduce framework called Disco [12], and a transactional DHT called Scalaris with Paxos support [33]. Philosophically, Erlang revolves around a notion of programming *concurrent processes*, rather than data. We do not have experience to share regarding the suitability of Erlang for datacenter programming. For the moment, Disco is significantly less functional than BOOM, lacking a distributed file system, multiple scheduling policies, and high availability via consensus. The Disco FAQ warns that "Hadoop is probably faster, more scalable, and more featureful" [12]. By contrast, BOOM performance is within a modest factor of Hadoop in apples-to-apples performance tests, and adds significant features. Erlang and Overlog seem to offer roughly similar code size. For example the Scalaris Paxos implementation in Erlang has more lines of code than our Overlog version, but in the same order of magnitude. In Section 7 we reflect on some benefits of a data-centric language, which may not be as natural in Erlang's process-centric model.

Distributed state machines are the traditional formal model for distributed system implemtations, and can be expressed in languages like Input/Output Automata (IOA) and the Temporal Logic of Actions (TLA) [28]. These ideas have been used extensively for network protocol design and verification [4, 7]. They also form the basis of the MACE [22] language for overlay networks. Although we were aware of these efforts, we were attracted to the notion of a data-centric language that would easily provide query-like facilities for monitoring and other datacenter management tasks. This seemed particularly useful for classical protocols such as Paxos that were originally specified in terms of logical invariants. As we discuss

```
path(@Start, End, X, Cost1+Cost2)
        :- link(@Start, X, Cost1),
           path(@X, End, Hop, Cost2);

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

Figure 1: Simple Overlog for computing paths from links, along with the analogous SQL statement.

in Section 6, this was confirmed by our experience, especially when we were able to convince ourselves of the correctness of our implementations by metaprogramming our logic.

Our use of metaprogrammed Overlog was heavily influenced by the metaprogrammed query optimization in P2's Evita Raced approach [11], and the security and typechecking features of Logic Blox' LBTrust [29]. Some of our monitoring tools were inspired by Singh et al [36], though our metaprogrammed implementation is much simpler than that of P2.

2. BACKGROUND

The Overlog language is sketched in a variety of papers. Originally presented as an event-driven language [27], it has evolved a more pure declarative semantics based in Datalog, the standard deductive query language from database theory [40]. Our Overlog is based on the description by Condie et al. [11]. We briefly review Datalog here, and the extensions presented by Overlog.

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:

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

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 col-lists of the body terms. For example, a canonical Datalog program for recursively computing paths from links [26] is shown in Figure 1 (ignoring the Overlog-specific `@` signs), along with analogous SQL. Note how the SQL `WHERE` clause corresponds to the repeated use of the variable `X` in the Datalog.

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. The Overlog data model consists of relational tables "horizontally" partitioned row-wise across a set of machines based on some column called the *location specifier*, which is denoted by the symbol `@`. A tuple is stored at the address specified in its location specifier column. JOL generalizes this slightly by supporting "local" tables that have no location specifier. This is a notational shorthand to prevent bugs: the same effect can be achieved by adding an additional location specifier column such tables, and ensuring that the value for each tuple is always "localhost". In Figure 1, the location specifiers reflect typical network routing tables, with each link or path stored at its source.

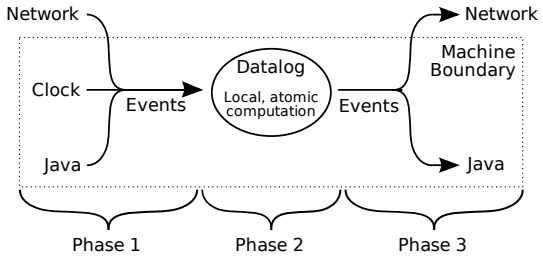

Figure 2: An Overlog timestep at a participating node: incoming events are applied to local state, a logical Datalog program is run to fixpoint, and any outgoing events are emitted.

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/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 [40], recursively evaluating the rules until no new results are generated. At the end of each local Datalog fixpoint, the local Datalog tables (including stored data and deductions) are consistent with each other via the rules. In the third phase, outbound events are sent over the network, or to local Java extension code (modeled as remote tables). Note that while Datalog is defined over static databases, the first and third phases allow Overlog programs to mutate state over time.

Communication in Overlog happens as a side-effect of data partitioning. Loo et al. show that any Overlog program can be compiled into a form where the body relations join on the same location-specifier variable, so that all relational processing is localized [26]. They also prove eventual consistency of the distributed tables under the rules, when certain simplifying assumptions hold. In Section 7 we discuss our experience with this model.

JOL is an Overlog runtime implemented in Java, based on a dataflow of operators similar to P2 [27]. JOL implements *metaprogramming* akin to P2's Evita Raced extension [11]: 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 [37]. 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 (reduce)

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.

2.1 Experimental Setup

We validated our results on a 101-node cluster on EC2. For Hadoop experiments, a single 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 "extra large" EC2 instance with 7.2 GB of memory and 8 virtual cores, with the equivalent of a 2.33GHz Intel Xeon processor per core. Our slave nodes executed on "medium" EC2 VMs with 1.7 GB of memory and 2 virtual cores, with the equivalent of a 2.33GHz Intel Xeon processor per core.

3. INITIAL BOOM 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 March, 2009. Refinement of JOL has been an ongoing effort, informed by the experience of writing BOOM. In Section 7 we reflect briefly on language and runtime lessons related to JOL.

We tried two design styles in developing the two halves of BOOM. For the MapReduce engine, we essentially "ported" the Hadoop master node's Java 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

In beginning our MapReduce port, we wanted 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 [45]. 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 developing the 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

The Hadoop MapReduce source code is based on the description of Google's implementation [13]. It has a single master called the JobTracker, which manages a number of workers called TaskTrackers. A job is divided into a set of map and reduce *tasks*. The JobTracker assigns map tasks to nodes; each task reads a 64MB *chunk* from the distributed file system, runs user-defined map code, 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. A TaskTracker running a reduce task fetches the corresponding hash buckets from all mappers, sorts locally by key, runs the reduce function and writes results into the distributed file system

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, dirty
taskTracker	TaskTracker definitions	name, hostname, state, map_count, reduce_count, max_map, max_reduce, dirty
trackerAction	Generated actions	tracker, action

Table 1: BOOM-MR tables and selected attributes defining the JobTracker state.

The Hadoop scheduler is part of the JobTracker. The scheduler multiplexes TaskTracker nodes across several jobs, executing maps and reduces concurrently. Each TaskTracker has a fixed number of slots for executing Map/Reduce 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. As in Google's paper, Hadoop will often schedule *speculative* tasks to reduce a job's response time by preempting "straggler" nodes [13]. Scheduling these speculative tasks is one topic of interest in recent work [45].

Our initial goal was to port the JobTracker code to Overlog. We began by identifying the key state maintained by the JobTracker, which is encapsulated in the *org.apache.hadoop.mapred* Java package. 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 five Overlog tables, shown in Table 1.

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 a field called *jobConf* that can hold a Java object constructed by legacy Hadoop code, which captures the configuration of the job. The *task* relation identifies each task in each job. The attributes for a task 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, under speculation or if the initial execution attempt failed. The *taskAttempt* relation maintains the state of each such attempt. In addition to a progress percentage $[0..1]$, 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 the input_loc.

The *taskTracker* relation identifies each TaskTracker in the cluster by a unique name. It is also used to constrain the scheduler, which can assign map and reduce tasks up to the max_map and max_reduce attributes of each tracker.

Having "table-ized" the JobTracker internals, we still had to translate from the traditional Java-based Hadoop APIs into this representation. For inbound messages, we stubbed out

the original Java message handlers to place tuples on the JOL event queue, where they are picked up by the JOL runtime. We chose to leave Hadoop's job configuration parsing untouched; it populates the jobConf field of the *job* table. We wrote our Overlog rules to place outbound messages into the *trackerAction* table of Table 1. We then modified Hadoop's Java heartbeat code to drain this table via a simple JOL call between timesteps, and send the corresponding API action to the TaskTracker mentioned in each *trackerAction* tuple.

The internal JobTracker Overlog rules maintain the bookkeeping of the internal tables based on inbound messages that are turned into *job*, *taskAttempt* and *taskTracker* tuples. This logic is largely straightforward, ensuring that the relations are internally consistent. Scheduling decisions are encoded in the *taskAttempt* table, which assigns tasks to TaskTrackers. These decisions are encapsulated in a set of policy rules we discuss next, and invoked via a join when *taskTracker* tuples exist with unassigned slots.

3.1.2 Scheduling Policies

MapReduce scheduling has been the subject of recent research, and one of our early motivations for building BOOM was to make that research extremely easy to carry out. With our initial BOOM-MR implementation in place, we were ready to evaluate whether we had made progress on that front. We had already implemented Hadoop's default First-Come-First-Served policy for task scheduling, which was captured in 9 rules (96 lines) of scheduler policy. To evaluate extensibility, we chose to replace that with the recently-proposed LATE policy [45], 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 [45], 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 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 [45]. We compute these thresholds via the five Overlog rules shown in Figure 3. 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.

The entire LATE policy can be applied to BOOM-MR's Overlog as a 82 line patchfile that adds a 64 line Overlog file and edits a single existing Overlog file. The Java LATE policy applied to vanilla Hadoop is an 800 line patchfile; about 200 lines of that file implement LATE, and the other approximately 600 lines modify 42 Java files in Hadoop.

```
// 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(TrackerName, JobId, Type, avg<PRate>) :-
    task(JobId, TaskId, Type, _),
    taskAttempt(JobId, TaskId, AttemptId, Progress,
                State, Phase, TrackerName, 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(TrackerName, JobId, Type, AvgPRate);

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

Figure 3: Overlog to compute statistics for LATE.

3.1.3 BOOM-MR Results

To compare the performance of our Overlog-based JobTracker with the stock version of the JobTracker, we used Hadoop version 18.1. Our workload was a wordcount on a 30 GB file. The wordcount job consisted of 481 map tasks and 100 reduce tasks. Figure 4 contains a grid of experiments performed on the EC2 setup described in Section 2.1. 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. Each graph reports a cumulative distribution of map and reduce task completion times (in seconds). The evolution of map task completion times occurs in three waves. This occurrence is due to the limited number of map tasks, in this case 200, that can be scheduled at any given time. On the other hand, all reduce tasks can be scheduled immediately. However, no reduce task will finish until all map tasks have completed since each reduce task requires the output of all map tasks.

The upper-left graph serves as a baseline for subsequent graphs by reporting the result of running our workload on Hadoop 18.1 over HDFS. The lower-left graph reports the result of running BOOM-MR over HDFS. The graph shows that map and reduce task completion times under BOOM-MR are nearly identical to Hadoop 18.1. We postpone the description of the upper-right and lower-right graphs to Section 3.2.

Figures 5 and 6 show the cumulative distribution of the

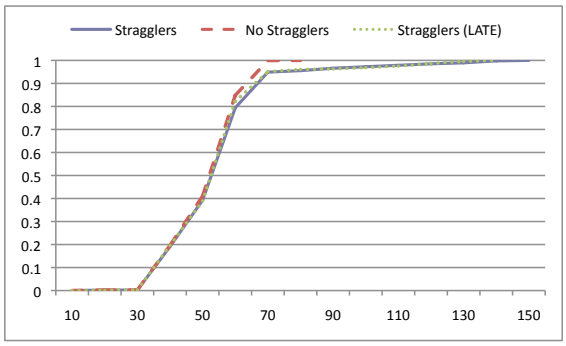

Figure 5: CDF of map task elapsed runtime (secs) with and without stragglers.

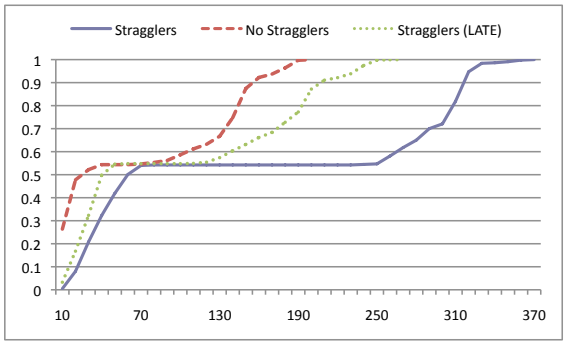

Figure 6: CDF of reduce task elapsed runtime (secs) with and without stragglers.

elapsed runtime for map and 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. 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. Figure 5 shows that the delay due to the loaded map stragglers only affects those six nodes. The reduce tasks are scheduled in two waves. The first wave of 200 reduce tasks is scheduled concurrently with the map tasks. This first wave of reduce tasks will not finish until all map tasks have completed, which increases the elapsed runtime of these tasks. The second wave of reduce tasks will not experience the delay due to map tasks since it is scheduled after all map tasks have finished. Figure 6 shows this effect, and also demonstrates how the LATE implementation in BOOM handles stragglers much more effectively than the default speculation policy ported from Hadoop. This echoes the results of Zaharia, et al. [45]

3.1.4 Discussion

Building BOOM-MR was the first major step in our implementation. We had an initial version running after a month of development, and have continued to tune it until very recently. The BOOM-MR codebase 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.

Our experience gutting Hadoop and inserting BOOM was not always pleasant. It took significant time to understand the Hadoop Java code and figure out what aspects of the API "skin" to keep. Given that we were committed to preserving the TaskTracker and client APIs, 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 regarding the number of input files and the requested number of output files.

In retrospect, it was handy to be able to draw the Java/Overlog boundaries flexibly. This kept us focused on porting the more interesting Hadoop logic into Overlog, while avoiding ports of relatively mechanical details — particularly as they related to the API. We also found that the Java/Overlog interfaces we had were both necessary and sufficient for our needs. We made use of essentially every possible interface: 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. However we did not identify an interface that was clearly missing from the system, or one that would have made our lives substantially easier.

With respect to the Overlog itself, we did find it much simpler to extend and modify than the original Hadoop code in Java. This was especially true with the scheduling policies. We have been experimenting with new scheduling policies recently, and it has been very easy to modify the 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 the BOOM-MR code 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. HDFS is somewhat more substantial. Although its metadata is still centralized, the actual data in HDFS is distributed and replicated [5]. HDFS is loosely based on GFS [16], and is targeted at storing large files for full-scan workloads.

In HDFS, file system metadata is stored at a centralized *NameNode*, while file data is partitioned into 64MB chunks and stored at a set of *DataNodes*. Each chunk is typically stored at three DataNodes to provide fault tolerance. The canonical record of which chunks are stored at which DataNode is not persistently stored at the NameNode; instead, DataNodes periodically send heartbeat messages to the NameNode containing the set of chunks stored at the DataNode. The NameNode keeps a cache of 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 each of the chunks stored at the crashed DataNode to ensure fault tolerance.

Clients contact the NameNode only to perform metadata

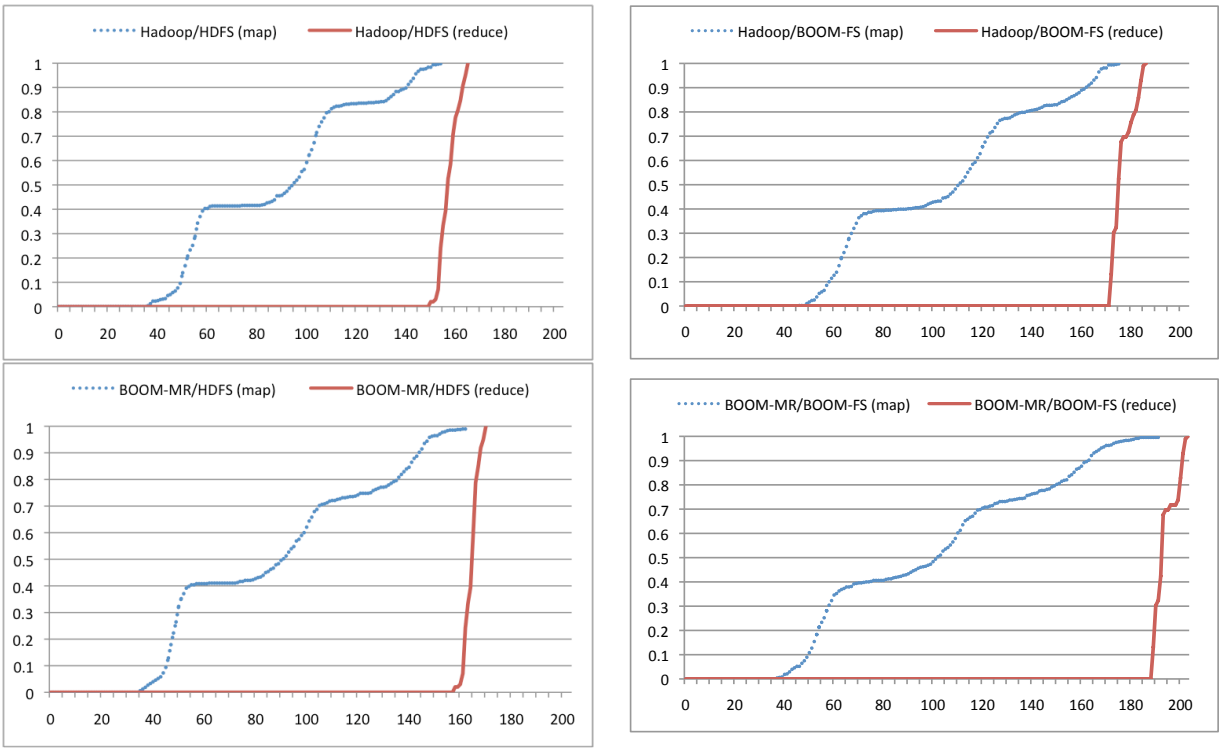

Figure 4: CDF of map and reduce task completion time (secs) running Hadoop and BOOM-MR over HDFS and BOOM-FS.

operations, such as obtaining the list of chunks in a file; all data operations involve only clients and DataNodes. Note that because of HDFS's intended workload, it 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: while for MapReduce we only rewrote the core JobTracker code, for BOOM-FS we chose to write code for the NameNode, DataNode, and client library interface. This allowed us to write the master-slave APIs in Overlog without replicating HDFS-internal APIs, and limited our Hadoop/Java compatibility task to implementing the Hadoop file system API, which we did by creating a simple translation layer between Java 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 an elegant separation of control and data paths: 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 relatively simple high-bandwidth data-path routines "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 achieved both elegance and high-performance.

3.2.2 File System State

The first step of our rewrite was to represent file system metadata as a collection of relations. Expressing file system policy then becomes a matter of writing queries over this schema, rather than creating algorithms that walk the file system's data structures. This takes the spirit of the recent work on declarative file system checking [19] one level deeper, to cover all file system metadata logic. A simplified version of the relational file system metadata in BOOM-FS is shown in Table 2.

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 on 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 crashed 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

[2]The order in which the chunks appear in a file must also be specified, because relations are unordered. In the current system, we assign chunk IDs in a monotonically increasing fashion and only support append operations, so the client can determine the order of chunks in a file by sorting the chunk IDs.

Name	*Description*	*Relevant attributes*
file	Files	fileid, parentfileid, name, isDir
fchunk	Chunks per file	chunkid, fileid
datanode	DataNode heartbeats	address, lastHeartbeatTime
hb_chunk	Chunk heartbeats	nodeAddr, chunkid, length

Table 2: Relations defining file system metadata.

```
// 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 7: Example Overlog rules over the file system metadata.

the Stasis storage library [34] to achieve durability, by writing the durable state modifications to disk as an atomic transaction at the end of each fixpoint. We return to a discussion of Stasis in Section 4.

Since a file system is naturally hierarchical, it is a good fit for a recursive query language like Overlog. For example, Figure 7 contains a single Overlog rule that infers fully-qualified pathnames from the parent information in the *file* relation. The *fqpath* relation defined by these rules allows the file ID associated with a given absolute path to be easily determined. Because this information is queried frequently, we configured the *fqpath* relation to be cached after being computed. JOL automatically updates the cache of *fqpath* correctly when its input relations change via *materialized view maintenance* logic [32]. For example, removing a directory `/x` will cause the *fqpath* entries for the children of `/x` to be removed. 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. Note that 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.

The state at each DataNode is simply the set of chunks stored by that node. The actual 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.

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. As illustrated by P2 [27], client-server message generation and handling patterns are easy to implement in Overlog. 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.

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

// Chunk exists => return success and set of addresses
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_chunks(ChunkId, _);
```

Figure 8: Overlog rules that return the set of DataNodes that hold a given chunk.

For each command in the metadata protocol, there is a single declarative 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 is shown for Chunk Location requests in Figure 8.

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. We believe that the history of concurrency control in relational databases suggests that high performance can be achieved while retaining a simple programming model.

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 Overlog `periodic` relation [27], whose tuples appear on the JOL event queue via a JOL runtime timer, rather than via network events. In addition, control protocol messages from the NameNode to DataNodes are deduced when conditions specified by certain rules indicate that system invariants are unmet; for example, when the number of replicas for a chunk drops below the specified 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 in Overlog 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 file server in Java that runs on

the DataNodes.

Our resulting initial BOOM-FS implementation has performance, scaling and failure-handling properties similar to those of HDFS. Figure 4 demonstrates that the performance of BOOM-FS follows the general trend of native HDFS, though currently it runs somewhat slower. Like HDFS, our implementation tolerates significant DataNode failure rates, but has a single point of failure and scalability bottleneck — at the NameNode.

3.2.4 Discussion

We began implementing BOOM-FS in September, 2008. Within two months 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, spread across a few debugging sessions, as BOOM was the first serious user of JOL's persistent tables. BOOM-FS consists of 85 Overlog rules in 469 lines of code, and 1431 lines of Java. The DataNode implementation accounts for 414 lines of Java; the remainder is mostly devoted to system configuration, bootstrapping, and a client library. The implementation of the Hadoop client library API required an additional 400 lines of Java. By contrast, HDFS is an order of magnitude larger: approximately 21,700 lines of Java.

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 sidesteps most distributed systems challenges regarding replication and fault-tolerance by implementing coordination in a centralized fashion at a single NameNode. As a result, most of our implementation consists of simple message handling and the management of a hierarchical namespace in a tabular representation. 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 a slower-to-recover warm standby scheme was posted to the Hadoop issue tracker in October of 2008 ([20] 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 [24]. It is also considered a challenging distributed algorithm to implement in a practical fashion [9], making it a natural choice for our experiment in distributed programming.

When we began working on availability, we had two reasons to believe that we could cleanly retrofit a hot standby solution into BOOM-FS. First, data-centric programming had already forced us to encode the relevant NameNode state into a small number of relational tables, so we knew what data we needed to replicate. Second, we were encouraged by a concise Overlog implementation of simple Paxos that had been achieved in an early version of P2 [38]. On the other hand, we were sobered by the fact that the Paxos-in-P2 effort failed to produce a useable implementation; like the Paxos implementation at Google [9], they discovered that Lamport's papers [25, 24] present just a sketch of what would be necessary to implement Paxos in a practical environment. The "Paxos Made Simple" paper [25], for example, leaves the details of executing more than a single round of consensus to the reader, and has no provisions for liveness (ensuring forward progress) or catch-up (logic to be carried out when a participant reconnects after failure).

We began by creating an Overlog implementation of basic Paxos akin to the P2 work [38], 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. Perhaps more helpful than the size was the fitness of Overlog to the task: our Overlog rules corresponded nearly line-for-line with the statements of invariants from Lamport's original paper [24]. Our entire implementation, commented with Lamport's invariants, 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.

Having reachieved that milestone, we needed to add the necessary extensions to convert basic Paxos into a working primitive for a distributed log. This includes 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 [9], 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

Number of NameNodes	Failure condition	Avg. Completion Time (secs)	Standard Deviation
1	None	101.89	12.12
3	None	102.7	9.53
3	Backup	100.1	9.94
3	Primary	148.47	13.94

Table 3: Completion times for BOOM jobs with a single NameNode, three Paxos-enabled NameNodes, failure of a backup NameNode, and failure of the primary NameNode.

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 that Paxos log table captures the globally-accepted order of actions on all replicas.

Finally, to implement Paxos reliably we had to add a disk persistence mechanism to JOL, a feature that was not considered in P2. We chose to use the Stasis storage library [34], which provides atomicity and durability for concurrent transactions, and handles physical consistency of its indexes and other internal structures. Unlike many transactional storage libraries, Stasis does not provide a default mechanism for concurrency control. This suited our purposes well, since Overlog's timestep semantics isolate the local database from network events, and take it from one consistent fixpoint to another. We implemented each JOL fixpoint as a single Stasis transaction.

Many of the tables in BOOM represent transient in-memory data structures, even though they are represented as "database" tables. Hence we decided to allow JOL programs to decide which tables should be durably stored in Stasis, and which should be transient and in-memory. Fixpoints that do not touch durable tables do not create Stasis transactions.

4.3 Evaluation

We had two goals in evaluating our availability implementation. At a fine-grained level, we wanted to ensure that our complete Paxos implementation was operating according to the specification in the papers. This required logging and analyzing network messages sent during the Paxos protocol. We postpone discussion of this issue to Section 6, where we present the infrastructure we built to help with debugging and monitoring. At a coarser grain, we wanted to see the availability feature "in action", and get a sense of how our implementation would respond to master failures.

For our first reliability experiment, we evaluated the impact of the consensus protocol on BOOM system performance, and the effect of failures on overall completion time. To this end, we ran a Hadoop wordcount job on a 5GB input file with a cluster of 20 machines, varying the number of master nodes and the failure condition. These results are summarized in Table 3. 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 9 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.

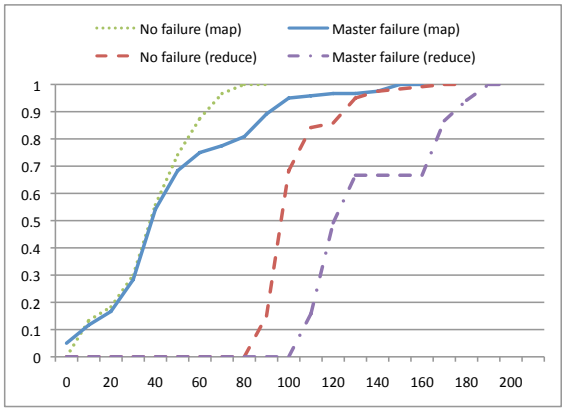

Figure 9: CDF of completed tasks over time (secs), with and without failures.

4.4 Discussion

The Availability revision was our first foray into what we consider 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, and the author later uses these invariants in his proof of correctness. Translating them 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. On the other hand, locking ourselves into the invariant-based implementation early on made adding enhancements and optimizations more difficult, as these were often specified as state machines in the literature. 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, special cases like this would either have resulted in duplication of rule logic, or explicit capturing of state. In certain cases we chose the latter, compromising somewhat our high-level approach to protocol specification. Avoiding races by allowing only one tuple to enter certain dataflows in each fixpoint also required a certain amount of lower-level reasoning that was not altogether natural to express in Overlog.

Our experience with managing persistence was as simple as we had hoped. Lamport's description of Paxos explicitly distinguishes between state that should be made durable, and state that should be transient; our implementation had already separated this state in separate relations, so we followed Lamport's description. Adding durability to BOOM-FS metadata was similar; the relations in Table 2 are marked durable, whereas "scratch tables" that we use to compute responses to file system requests are transient. These are simple uses of Overlog persistence, but for our purposes to date this model has been

sufficient. Stasis seems to be an elegant fit due to its separation of atomic, durable writes from traditional transactions locking for isolation and consistency management. However, given the modest load we put on Stasis, we also would probably have been fine using a traditional database.

5. THE SCALABILITY REV

HDFS NameNodes manage large amounts of in-memory metadata for filenames and file chunk lists. The original GFS paper acknowledged that this could cause significant memory-pressure [16], but at the time the authors suggested solving the problem by purchasing more RAM. More recently we have heard from contacts at large Internet services that NameNode scaling can in fact be an issue in many cases. We decided it would be interesting to try to scale out our NameNode implementation by partitioning its metadata across multiple *NameNode-partitions*. From a database design perspective this seemed trivial — it involved adding a "partition" column to some Overlog schemas. To keep various metadata tasks working, we added partitioning and broadcast support to the client library. The resulting code composes cleanly with our availability implementation; each NameNode-partition can be a single node, or a Paxos quorum of replicas.

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 chose 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 HDFS 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

Scaling up the NameNode was every bit as simple as we might have hoped. It took 8 hours of developer time to implement our NameNode partitioning; two of these hours were spent adding partitioning and broadcast support to the Overlog code. This was a clear benefit of the data-centric approach: by isolating the system state into simple relational tables, it became a textbook exercise to partition that state across nodes.

The simplicity of partitioning also made it easy to think through its integration with reliability. We had originally been concerned about the complexity of implementing partitioned quorums. We found that our simplified semantics were sufficient to support MapReduce behavior though we believe that adding file renames via two-phase commit should be easy as well. Our confidence in being able to compose techniques from the literature is a function of the simplicity and compactness of our code. Although much of the time spent implementing partitioning was dedicated to debugging, most of this time was spent dealing with defects in the Java portion of our client library. In contrast, the relevant Overlog code fit on a single screen and we quickly became confident of its correctness. In retrospect, moving more client functionality into Overlog would have simplified our design and saved debugging effort.

6. THE MONITORING REV

As our BOOM 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 the Overlog language is well-suited to writing distributed monitoring queries, and offers a naturally introspective approach: simple Overlog queries can monitor complex Overlog protocols [36]. Following that idea, we decided to develop a suite of debugging and monitoring tools for our own use.

While Singh et al. implemented their introspection via custom hooks in the P2 dataflow runtime, we were able to implement them much more simply via Overlog program rewriting. This was made easy for us because of the metaprogramming approach that we adopted from Evita Raced [11], which enables Overlog rewrites to be written at a high level in Overlog. This kept the JOL runtime lean, and allowed us to prototype and evolve these tools very quickly.

6.1 Invariants

One advantage of a logic-oriented language like Overlog is that it encourages the specification of system invariants — in fact, local rules in Overlog are essentially invariant specifications over derived relations. It is often useful to add additional "watchdog" invariants to debug an Overlog program, even though they are not required during execution. For example, one can count that the number of messages sent by a protocol like Paxos is as expected. Formally, this may be implicit in the protocol specification, and hence redundant. But the redundancy only holds if the protocol is correctly specified; the point of adding such logic is to ensure that the specification is correct. Moreover, distributed rules in Overlog cause asynchronous behavior that happens across timesteps and a network, and such rules are only *attempts* to achieve invariants. An Overlog program needs to be enhanced with global coordination mechanisms like two-phase commit or distributed snapshots to convert distributed Overlog rules into global invariants [10]. Singh et al. have shown how to implement Chandy-Lamport distributed snapshots in Overlog [36]; 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 tend to improve readability in addition to reliability. This is important in a language like Overlog: it is a terse language (too terse, in our opinion), and its 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 little help in understanding the issues. 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 hand-replicate the body of each rule in an Overlog program, and send its outputs to a logging predicate, which can have a local or remote location specifier. 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 lead to an additional pair of rules:

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

trace_precondition(@Master, "paxos", Predicate,
                   "quorum1", Tstamp) :-
    priestCnt(@Master, Pcnt),
    lastPromiseCnt(@Master, Round, Vcnt),
    Vcnt > (Pcnt / 2),
    Predicate := "quorum",
    Tstamp := System.currentTimeMillis();
```

This approach captures the dataflow generated per rule in a trace relation that can be queried later. At a finer level of detail, the prefixes of rule bodies can be replicated and logged/counted in a similar fashion, capturing the dataflow emerging from every step of a rule's deduction. For example, the previous rule would lead to:

```
tap_priestCnt(@Master, Pcnt) :-
    priestCnt(@Master, Pcnt);

tap_lastPromiseCnt@Master, Round, Vcnt) :-
    lastPromiseCnt(@Master, Round, Vcnt);

tap_quorum_0(@Master, Round) :-
    tap_priestCnt(@Master, Pcnt),
    tap_lastPromiseCnt(@Master, Round, Vcnt);

tap_quorum(@Master, Round) :-
    tap_priestCnt(@Master, Pcnt),
    tap_lastPromiseCnt(@Master, Round, Vcnt),
    Vcnt > (Pcnt / 2);
```

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 running in profiling mode, this overhead is often acceptable. But writing the new trace rules by hand is tedious.

Using the metaprogramming approach of Evita Raced, we were able to write a *trace rewriting* program in Overlog in terms over 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.[3]

Network traces fall out of this approach naturally as well. Any dataflow transition that results in network communication can be named as such during trace rewriting.

Using this idea, it took us less than a day to create a 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 in every participating node, approximately 12 summary rules that can be run in a centralized location, and several hundred lines of Java glue. We ran our regression tests through this tool, and immediately found both "dead code" rules in our BOOM programs, and code that we knew needed to be exercised by the tests but was as-yet uncovered.

The results of this tracing rewrite can be used for further analysis in combination with other parts of our management infrastructure. For example, using the assertion style described above, we created unit tests corresponding to the correctness goal mentioned in Section 4.3: we confirmed that the message complexity of our Multi-Paxos implementation was exactly as predicted by the algorithm, both at steady state and under churn. This gave us more confidence that our Multi-Paxos implementation was correct.

6.3 Logs

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 [6]. 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.

To get started, we implemented Java modules to produce performance counters by reading the `/proc` file system, logging the counters 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 im-

[3] A query optimizer can reorder the terms in the body of a rule, so the trace rewriting needs to happen after that query optimization. The Evita Raced architecture provides a simple "staging" specification to enforce this.

plement 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 data 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 BOOM implementation began in earnest just six months before this paper was written, and involved only four developers. Our overall experience has been enormously positive — we are frankly surprised at our own programming 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 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 the LATE scheduler thresholds, 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 simple *interposition* [21] of components in a natural dataflow manner: the dataflow "pipe" between two system modules could be easily rerouted to go through a third module. (Syntactically, this is done in Overlog by interposing the input and output tables of the third module into rules that originally joined the first two modules' output and input.) This enabled the simplicity of incorporating our Overlog LATE scheduler into BOOM-MR (Section 3). Because dataflows can be routed across the network (via the location specifier in a rule's head), interposition can involve distributed logic as well — this is how we easily put Paxos support into the BOOM-FS NameNode (Section 4).

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 multithreaded 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 piping Map and Reduce tasks. Candidly, however, we have relatively limited experience with event-loop programming, and the compactness of languages like Erlang suggests that it is a plausible alternative to the data-centric approach in this regard. On the hand, we do have experience writing multi-threaded code with locking, and 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 code, and we view this as a clear victory for the programming model, however it arose.

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 issue is worth commenting on briefly — if only to sympathize with our readers. We have grown to dislike the Datalog convention that connects columns across relations via "unification": repetition of variable names in different positions. We found that it makes Datalog harder than necessary to write, and even harder to read. A text editor with proper code-coloring helps to some extent, but we suspect that no programming language will grow popular with a syntax based on this convention. That said, the issue is eminently fixable: SQL's named-field approach is one option, and we can imagine others. In the end, our irritability with Datalog syntax was swamped 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). We had two reactions to the issue. The first is that the porting task did not actually turn out to be that hard: in most cases it amounted to writing message-handler style 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 a deeper understanding of the ideas involved.

With respect to consistency of storage, we were comfortable with our model of local storage transactions per fixpoint evaluation. 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 to compete with Hadoop, but we are conscious that it needs to improve. For example, we observed anecdotally that system load averages were much lower with Hadoop than with BOOM. One concrete step we plan to take in future is to implement an Overlog-to-C compiler; we believe we could reuse a large fraction of JOL to this end.

In the interim, we actually think the modest performance of the current JOL interpreter guided us to reasonably good design choices. By defaulting to traditional 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 conceptual efforts like developing Multi-Paxos.

8. CONCLUSION

We implemented BOOM 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 in recent months 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 incarnation 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.

9. REFERENCES

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing jalapeño in java. In *OOPSLA*, 1999.

[2] Amazon Corp. Amazon Simple Storage Service (Amazon S3), Feb. 2009. `http://aws.amazon.com/s3/`.

[3] M. P. Ashley-Rollman, M. De Rosa, S. S. Srinivasa, P. Pillai, S. C. Goldstein, and J. D. Campbell. Declarative Programming for Modular Robots. In *Workshop on Self-Reconfigurable Robots/Systems and Applications*, 2007.

[4] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks*, 14, 1987.

[5] D. Borthakur. HDFS architecture, 2009. `http://hadoop.apache.org/core/docs/current/hdfs_design.html`.

[6] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa, a large-scale monitoring system. In *Cloud Computing and its Applications*, pages 1–5, Chicago, IL, October 2008.

[7] S. Budkowski and P. Dembinski. An introduction to Estelle: A specification language for distributed systems. *Computer Networks*, 14, 1987.

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

[9] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407, 2007.

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

[11] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. In *VLDB*, Aug. 2008.

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

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

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, Oct. 2007.

[15] J. Eisner, E. Goldlust, and N. A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP)*, 2005.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[17] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[18] Greenplum. A unified engine for RDBMS and MapReduce, 2009. `http://www.greenplum.com/resources/mapreduce/`.

[19] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *OSDI*, 2008.

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

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

[22] C. Killian, J. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language support for building distributed systems. In *PLDI*, 2007.

[23] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-Sensitive Program Analysis as Database Queries. In *PODS*, 2005.

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

[25] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[26] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *SIGMOD*, pages 97–108, 2006.

[27] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

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

[29] W. R. Marczak, D. Zook, W. Zhou, M. Aref, and B. T. Loo. Declarative reconfigurable trust management. In *CIDR*, 2009.

[30] Microsoft Corp. Services – SQL Data Services | Azure Services Platform, Feb. 2009. `http://www.microsoft.com/azure/data.mspx`.

[31] O. O'Malley. Hadoop map/reduce architecture, July 2006. Presentation, `http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/HadoopMapReduceArch.pdf`.

[32] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3 edition, 2002.

[33] T. Schutt, F. Schintke, and A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. In *SIGPLAN Workshop on Erlang*, 2008.

[34] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *OSDI*, pages 29–44, 2006.

[35] A. Singh. Aster nCluster in-database MapReduce: Deriving deep insights from large datasets, 2009. `http://www.asterdata.com/resources/downloads/whitepapers/Aster_MapReduce_Technical_Whitepaper.pdf`.

[36] A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *EuroSys*, pages 389–402, 2006.

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

[38] B. Szekely and E. Torres, Dec. 2005. Harvard University class project, `http://www.klinewoods.com/papers/p2paxos.pdf`.

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

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

[41] J. Whaley. Joeq: A virtual machine and compiler infrastructure. In *IVME*, June 2003.

[42] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *SIGMOD*, 2007.

[43] F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *ICDE*, 2006.

[44] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[45] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.