BOOM: Data-Centric Programming in the Datacenter

Peter Alvaro
UC Berkeley
palvaro@eecs.berkeley.edu

Tyson Condie
UC Berkeley
tcondie@eecs.berkeley.edu

Neil Conway
UC Berkeley
nrc@eecs.berkeley.edu

Khaled Elmeleegy
Yahoo! Research
khaled@yahoo-inc.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@eecs.berkeley.edu

Russell Sears
UC Berkeley
sears@eecs.berkeley.edu

Abstract

The efficiencies of cloud computing enable a wide range of developers to access the power of large clusters. Unfortunately, parallel and distributed programming remains too complex for many of these developers, and slows the progress of even sophisticated distributed system builders. We conjecture that a broad range of distributed software can be recast naturally in a data-parallel programming model. We argue that this significantly raises the level of abstraction for programmers, improving code simplicity, speed of development, ease of software evolution, and program correctness. To evaluate these claims, the bulk of the paper presents our experience using the Overlog language to implement a "Big Data" analytics stack that is API-compatible with Hadoop and HDFS, providing comparable performance. We extended the system with complex distributed features not yet available in Hadoop, including availability, scalability, and unique monitoring and debugging facilities. We present both quantitative and anecdotal results from our experience, showing that a data-centric approach can substantially simplify distributed systems programming.

1. Introduction

Clusters of commodity hardware have become a standard architecture for datacenters over the last decade. The advent of *cloud computing* promises to commoditize this architecture, enabling third-party developers to simply and economically build and host applications on managed clusters.

[Copyright notice will appear here once 'preprint' option is removed.]

Today's cloud interfaces are especially convenient for launching multiple independent instances of traditional single-node services. But writing truly distributed software remains a significant challenge. Developers who wish to build distributed applications and infrastructure are still required to manage communication channels between pairs of machines, and to develop protocols to coordinate computation in an application-specific manner in the face of delays and failures. The difficulty of writing robust distributed code prevents most programmers from making innovative use of the distributed potential of cloud platforms.

Although distributed programming remains hard, one important subclass is relatively well-understood by programmers: data-parallel computations expressed using interfaces like MapReduce [10], Dryad [16], and SQL. These languages and tools significantly raise the programming abstraction for distributed systems. They mask the coordination of threads and events, and instead ask programmers to focus on functional or logical expressions over collections of data. These expressions are then auto-parallelized via a dataflow runtime that partitions and shuffles the data across machines in the network. But these programming models have traditionally been restricted to data analysis tasks—a rather specialized subset of distributed and parallel computing.

Our work in this paper begins with the conjecture that *the majority of distributed systems software can be mapped cleanly into a high-level, data-centric programming model*. If true, this suggests that tricky distributed programming tasks can be recast as much simpler distributed data processing tasks. This conjecture is strengthened by a review of recent literature on datacenter infrastructure (e.g., [5, 10, 11, 13]). Most of the logic in these efforts involves managing various forms of asynchronously-updated state including sessions, protocols, and storage. Few or none of the ideas in these systems involve intricate, uninterrupted sequences of computational steps.

If complex distributed programs can be cast as partitioned data-parallel computations, it should be possible to develop, debug and extend this code at a far higher level of abstraction than is used in current practice. And this should in turn provide significant improvements in code simplicity, speed of development, ease of system evolution, and program correctness.

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] One long-term goal of BOOM is to design a new programming language for distributed systems, but before embarking on that effort we wanted to ground ourselves in the experience of building a significant system in an existing language. For this first phase of the BOOM effort, we used the Overlog logic language, originally developed for Declarative Networking [22]. We chose Overlog both because of local familiarity in our group, and because its original application domain—peer-to-peer overlay networks—had some similarities to our datacenter setting.

As a first significant effort in data-centric distributed software development, we chose to build *BOOM Analytics*: an API-compliant reimplementation of the HDFS distributed filesystem and the Hadoop MapReduce engine. We named these two components *BOOM-FS* and *BOOM-MR*, respectively. In writing BOOM Analytics, we preserved the Java API "skin" of HDFS and Hadoop, but replaced complex internal state with a set of relations. This enabled us to express the system logic in a declarative language like Overlog.

The Hadoop stack appealed to us as a challenge for two reasons. First, it exercises the distributed power of a cluster. Unlike a farm of independent web service instances, the HDFS and Hadoop code entails coordination of large numbers of nodes toward common tasks. Second, Hadoop is a work in progress, and is still missing significant distributed systems features like availability and scalability of master nodes. We expected that developing these complex features would challenge our main conjecture, and that integrating them into our code would test the ease of evolution of a data-centric implementation.

1.2 Contributions

The bulk of this paper describes our experience implementing and evolving BOOM Analytics in Overlog, and running it on Amazon EC2. To our knowledge, BOOM Analytics is the largest and most complex application ever written in Overlog; we are not aware of another effort that is comparable in size and complexity to a production-quality distributed system. We document the relative ease of developing BOOM Analytics in Overlog, and the way that its data-centric design enabled us to rapidly introduce complex extensions, including Paxos-supported replicated-master availability, and multi-master state-partitioned scalability. We describe how the data-centric programming style facilitated debugging, and how by metaprogramming Overlog we were able to easily instrument our distributed system at runtime. We show that the performance of our resulting code is competitive with the original Hadoop codebase.

In describing our experience, we attempt to provide quantitative evidence for the software engineering and architectural benefits of data-centric distributed programming. As we describe each module of BOOM Analytics, we report the person-hours we spent implementing it, and the size of our implementation in lines of code (comparing against the relevant feature of Hadoop, if appropriate). Clearly, these are noisy metrics. But in most cases we are able to observe benefits that we believe transcend that noise: for example, order-of-magnitude reductions in code size. These quantitative distinctions are not intended as careful calibrations, but rather as evidence of significant qualitative changes in programmer efficiency.

We present the evolution of BOOM Analytics from a straightforward reimplementation of HDFS and Hadoop to a significantly enhanced system. We describe how our initial BOOM-FS prototype went through a series of major revisions ("revs") focused on *availability* (Section 4), *scalability* (Section 5), and *debugging and monitoring* (Section 6). We then detail how we designed BOOM-MR, and replaced Hadoop's task scheduling logic with a declarative scheduling framework implemented in Overlog (Section 7). In each case, we discuss how the data-centric approach influenced our design, and how the modifications involved were simple and well isolated from the earlier revisions. We compare the performance of BOOM Analytics with stock Hadoop in Section 8, and reflect on the lessons we learned in Section 9.

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 [10] has popularized functional dataflow programming with new audiences in computing. Also, a surprising breadth of recent research projects have proposed and prototyped declarative languages, including overlay networks [22], three-tier web services [33], natural language processing [12], modular robotics [4], video games [32], file system metadata analysis [15], and compiler analysis [18].

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 [32] — 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

[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.

techniques developed for declarative languages [32]. Declarative and dataflow languages can also share the same runtime, as demonstrated by recent integrations of MapReduce and SQL in Hive [30], DryadLINQ [34], 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 [25] and a transactional DHT called Scalaris with Paxos support [26]. However, neither of these efforts is as substantial as BOOM Analytics: for example, Disco currently lacks a distributed file system, and does not attempt to provide multiple scheduling policies or high availability via consensus. Philosophically, Erlang revolves around *concurrent actors*, rather than data. Experience papers regarding Erlang can be found in the literature (e.g., [6]). We focus here on a significant experience building distributed systems in a data-centric fashion. A closer comparison of actor-oriented and data-centric pros and cons is beyond the scope of this paper, but an interesting topic for future work.

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) [23]. 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.

Our use of metaprogrammed Overlog was heavily influenced by the Evita Raced Overlog metacompiler [9], and the security and typechecking features of Logic Blox' LB-Trust [24]. Some of our monitoring tools were inspired by Singh et al. [28], 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 [22], it has evolved a more pure declarative semantics based in Datalog, the standard deductive query language from database theory [31]. Our Overlog is based on the description by Condie et al. [9]. 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 *views*. A Datalog rule has the form:

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

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 [21] is shown in Figure 1 (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 1. Example Overlog for computing paths from links, along with an SQL translation of the second rule.

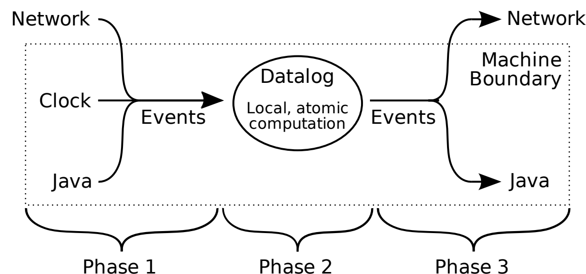

Figure 2. An Overlog timestep at a participating node: incoming events are applied to local state, the local Datalog program is run to fixpoint, and any outgoing events are emitted.

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 `@`.

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 [31], 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.

The original Overlog implementation (*P2*) is aging and targeted at network protocols, so we developed a new Java-based Overlog runtime we call *JOL*. JOL implements *metaprogramming* akin to P2's Evita Raced extension [9]: 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 [29]. 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.

We chose to develop a new Overlog runtime because we anticipated the need for tight integration with Java code, and because we were not interested in the flexible but complicated componentized transport capabilities of P2 [22]. This decision paid off, as we found the Java integration supported by JOL to be very useful while building BOOM Analytics. To simplify parser development, JOL uses the PEG-based Rats! parser generator [14], rather than the context-free grammar used by P2. JOL is otherwise very similar to P2 in its architecture.

3. HDFS Rewrite

Our first effort in developing BOOM Analytics was BOOM-FS, a clean-slate rewrite of HDFS in Overlog. HDFS is loosely based on GFS [13], 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.

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 1. BOOM-FS relations defining file system metadata. The underlined attributes in the table together make up the primary key of each relation.

Like GFS, HDFS maintains a clean separation of control and data protocols: metadata operations, chunk placement and DataNode liveness are decoupled from the code that performs bulk data transfers. This made our rewriting task straightforward. 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. By leveraging the Java integration features of JOL, we were able to use the most appropriate language for the problem at hand. As we reflect in Section 9, this produced a hybrid system that is both clean and efficient.

3.1 File System State

The first step of our rewrite was to represent file system metadata as a collection of relations (Table 1). We then implemented file system operations by writing queries over this schema.

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; each Overlog fixpoint brings the system from one consistent state to another. We used the Stasis storage library [27] to write 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 1 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

[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.

file system directory structures. 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 3. 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 relational view maintenance logic [31]. 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 be changed via simple Overlog table definition statements, without changing the semantics of the program. 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.

3.2 Communication Protocols

Both HDFS and BOOM-FS use three different protocols: the *metadata protocol* that clients and NameNodes use to exchange file metadata, the *heartbeat protocol* that DataNodes use to notify the NameNode about chunk locations and DataNode liveness, and the *data protocol* that 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.

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 [22], 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

```
// 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 3. Example Overlog for computing fully-qualified pathnames from the base file system metadata in BOOM-FS.

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

Table 2. Code size of two file system implementations.

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.

3.3 Discussion

After four person-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 two weeks. Adding metadata durability took about a day. 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. As Table 2 shows, BOOM-FS contains an order of magnitude less code than HDFS.

Like HDFS, the initial BOOM-FS prototype can tolerate DataNode failures but has a single point of failure and scalability bottleneck at the NameNode. We discuss how we improved NameNode fault tolerance and scalability in Sections 4 and 5, respectively. As we discuss in Section 8, the performance of BOOM-FS is competitive with HDFS.

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 scan-

ning 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 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 ([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 [19].

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 [19]. 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, which we further describe in [3].

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 [7], 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 9.

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 the technical report [2].

4.3 Discussion

In total, our Paxos implementation constituted roughly 400 lines of code and required six person-weeks of development time. Adding Paxos support to BOOM-FS took one person-week and required modifying ten BOOM-FS rules.

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 [13], and NameNode scaling is often an issue in practice at Yahoo!. Given the data-centric nature of BOOM-FS, we hoped to 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 [28]. 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 enhanced with global coordination mechanisms like two-phase commit or distributed snapshots to convert distributed Overlog rules into global invariants [8]. Singh et al. have shown how to implement Chandy-Lamport distributed snapshots in Overlog [28]; 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 prototype of 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 [9], 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.

7. MapReduce Port

In contrast to our clean-slate strategy for implementing BOOM-FS, our MapReduce implementation, BOOM-MR, involved refactoring Hadoop's core scheduling component into a declarative specification. Our goal in building BOOM-MR was to explore the idea of embedding a data-centric rewrite of a non-trivial component into an existing procedural system. MapReduce scheduling policies were one issue that had been treated in recent literature [35]. 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.

7.1 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 file *chunk* from the distributed file system, runs a user-defined map function, and partitions output key/value pairs into hash-buckets on local disk. Reduce tasks are created for each hash value and are coscheduled with map tasks. 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. Also, Hadoop will attempt to schedule *speculative* tasks to reduce a job's response time if it detects "straggler" nodes [10].

7.2 MapReduce Scheduling 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 3.

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

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 3. BOOM-MR relations defining JobTracker state.

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 simply a set of rules that join against the *taskTracker* relation to find TaskTrackers with unassigned slots, and schedules tasks by inserting tuples into *taskAttempt*. This architecture makes it easy for new scheduling policies to be defined. We describe our experiences implementing the LATE scheduling policy [35] in Section 7.3.

7.3 Evaluation

To validate the extensible scheduling architecture described in Section 7.2, we implemented both Hadoop's default First-Come-First-Serve (FCFS) policy and the recently-proposed LATE policy [35]. Our goals were both to evaluate the difficulty of building a new policy, and to confirm the faithfulness of our Overlog-based JobTracker to the Hadoop JobTracker using two different scheduling algorithms.

Implementing the default FCFS policy required 9 rules (96 lines of code). Implementing the LATE policy required 5 additional Overlog rules (30 lines of code). In comparison, LATE is specified in the paper via just three lines of pseudocode, but implementing the policy in vanilla Hadoop required adding or modifying an order of magnitude more (800) lines of Java. Further details of our LATE implementation can be found in the technical report [2].

We now compare the behavior of our LATE implementation with the results observed by Zaharia et al. using Hadoop MapReduce. We used 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. Each DataNode was configured to support executing up to 2 map tasks and 2 reduce tasks simultaneously. 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.

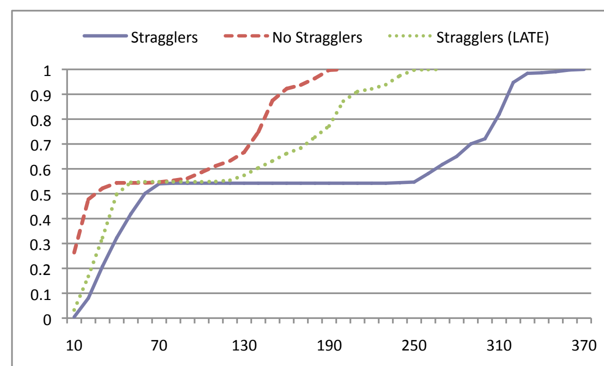

Figure 4. CDF of reduce task completion times (secs), with and without stragglers.

LATE focuses on how to improve job completion time by reducing the impact of "straggler" tasks. To simulate stragglers, we used the same EC2 cluster described above, except that we artificially placed additional load on six nodes. We ran a wordcount job on 30 GB of data, using 481 map tasks and 400 reduce tasks (which produced two distinct "waves" of reduces). Figure 4 shows the reduce task completion CDF for three different configurations. The plot labeled "No Stragglers" represents normal load, while the "Stragglers" and "Stragglers (LATE)" plots describe performance in the presence in stragglers using the default FCFS policy and the LATE policy, respectively. We omit map task completion from the CDF, because the artificial load had very little effect on map task execution—it just resulted in slightly 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 4 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. [35]

7.4 Discussion

The initial version of BOOM-MR required one person-month of development time. We spent an additional two person-months debugging and tuning BOOM-MR's performance for large jobs. BOOM-MR consists of 55 Overlog rules in 396 lines of code, and 1269 lines of Java. BOOM-MR is based on Hadoop version 18.1; we estimate that we removed 6,573 lines from Hadoop (out of 88,864). The removed code contained the core scheduling logic and the data structures that represent the components listed in Table 3. The Overlog patch that replaces the original Hadoop scheduler contains an order of magnitude fewer lines of code. The performance of BOOM-MR is very similar to that of Hadoop MapReduce, as we discuss in Section 8.

For this "porting" exercise, 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. For example, we chose to leave the data representation of the *jobConf* as a Java object rather than flatten it into a relation because it had no effect on the scheduling logic.

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. Informally, our Overlog code seems about as simple as the task should require: the coordination of MapReduce tasks is an elegantly simple design, and we feel that the simplicity of BOOM-MR is appropriate to the simplicity of the system's logic.

8. Performance Validation

While improved performance was not a goal of our work, we wanted to ensure that the performance of BOOM Analytics was competitive with Hadoop. We compared BOOM Analytics with Hadoop 18.1, using the 101-node EC2 cluster described in Section 7.3. The workload was a wordcount job on a 30 GB file, using 481 map tasks and 100 reduce tasks.

Figure 5 contains four 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 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.

9. 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 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.

9.1 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* [17] 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 7.2). 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 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,

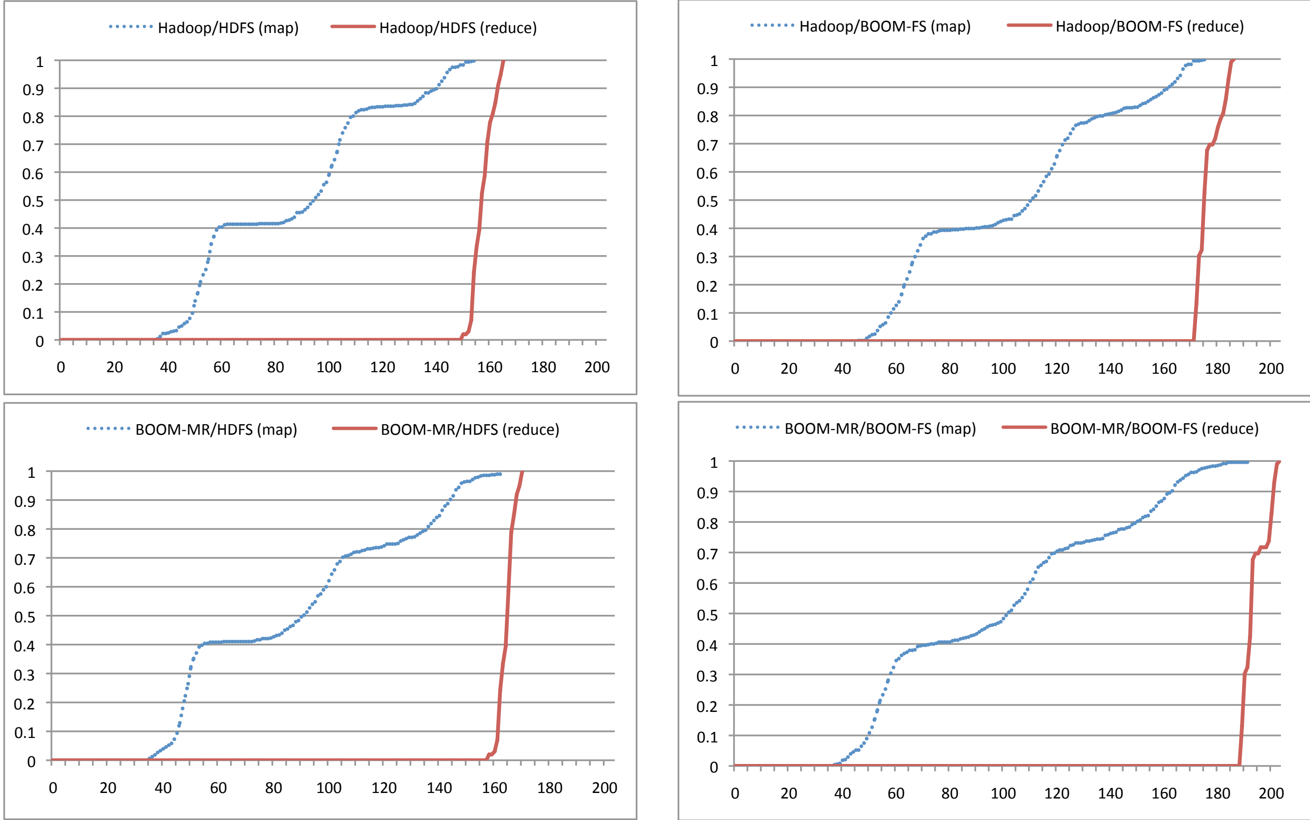

Figure 5. CDF of map and reduce task completion for Hadoop and BOOM-MR over HDFS and BOOM-FS. In all graphs, the horizontal axis is elapsed time in seconds, and the vertical represents % of tasks completed.

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.

9.2 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.

9.3 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.

10. 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.

References

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

[2] P. Alvaro et al. BOOM: Data-centric programming in the datacenter. Technical Report UCB/EECS-2009-113, EECS Department, University of California, Berkeley, Jul 2009.

[3] P. Alvaro et al. I Do Declare: Consensus in a logic language. In *NetDB*, 2009.

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

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

[6] D. Cabrero et al. ARMISTICE: an experience developing management software with Erlang. In *Proc. ACM SIGPLAN workshop on Erlang*, 2003.

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

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

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

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

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

[12] J. Eisner et al. Dyna: a declarative language for implementing dynamic programs. In *Proc. ACL*, 2004.

[13] S. Ghemawat et al. The Google file system. In *SOSP*, 2003.

[14] R. Grimm. Better extensibility through modular syntax. In *PLDI*, 2006.

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

[16] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. 1990.

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

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

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

[35] M. Zaharia, R. K. A. Konwinski, A.D. Joseph, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.