BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud

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

Building and debugging distributed software remains extremely difficult. We conjecture that by adopting a *data-centric* approach to system design and by employing *declarative* programming languages, a broad range of distributed software can be recast naturally in a data-parallel programming model. Our hope is that this model can significantly raise the level of abstraction for programmers, improving code simplicity, speed of development, ease of software evolution, and program correctness.

This paper presents our experience with an initial large-scale experiment in this direction. First, we used the Overlog language to implement a "Big Data" analytics stack that is API-compatible with Hadoop and HDFS and provides comparable performance. Second, we extended the system with complex distributed features not yet available in Hadoop, including high availability, scalability, and unique monitoring and debugging facilities. We present both quantitative and anecdotal results from our experience, providing some concrete evidence that both data-centric design and declarative languages can substantially simplify distributed systems programming.

Categories and Subject Descriptors H.3.4 [*Information Storage and Retrieval*]: Systems and Software—Distributed systems

General Terms Design, Experimentation, Languages

Keywords Cloud Computing, Datalog, MapReduce

EuroSys'10, April 13–16, 2010, Paris, France.

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.

Today's cloud interfaces are convenient for launching multiple independent instances of traditional single-node services, but writing truly distributed software remains a significant challenge. Distributed applications still require a developer to orchestrate concurrent computation and communication across machines, in a manner that is robust to delays and failures. Writing and debugging such code is difficult even for experienced infrastructure programmers, and drives away many creative software designers who might otherwise have innovative uses for cloud computing platforms.

Although distributed programming remains hard today, one important subclass is relatively well-understood by programmers: data-parallel computations expressed using interfaces like MapReduce [11], Dryad [17], and SQL. These programming models substantially raise the level of abstraction for programmers: they mask the coordination of threads and events, and instead ask programmers to focus on applying functional or logical expressions to collections of data. These expressions are then auto-parallelized via a dataflow runtime that partitions and shuffles the data across machines in the network. Although easy to learn, these programming models have traditionally been restricted to batch-oriented computations and data analysis tasks — a rather specialized subset of distributed and parallel computing.

We have recently begun the *BOOM* (Berkeley Orders of Magnitude) research project, which aims to enable developers to build orders-of-magnitude more scalable software using orders-of-magnitude less code than the state of the art. We began this project with two hypotheses:

1. Distributed systems benefit substantially from a *data-centric* design style that focuses the programmer's attention on carefully capturing all the important state of the system as a family of collections (sets, relations, streams, etc.) Given such a model, the state of the system can be distributed naturally and flexibly across nodes via familiar mechanisms like partitioning and replication.
2. The key behaviors of such systems can be naturally implemented using *declarative* programming languages that manipulate these collections, abstracting the programmer from both the physical layout of the data and the fine-grained orchestration of data manipulation.

Taken together, these hypotheses suggest that traditionally difficult distributed programming tasks can be recast as data processing problems that are easy to reason about in a distributed setting and expressible in a high-level language. In turn, this should provide significant reductions in code complexity and development overhead, and improve system evolution and program correctness. We also conjecture that these hypotheses, taken separately, can offer design guidelines useful in a wide variety of programming models.

1.1 BOOM Analytics

We decided to begin the BOOM project with an experiment in construction, by implementing a substantial piece of distributed software in a data-centric, declarative style. Upon review of recent literature on datacenter infrastructure (e.g., [7, 11, 12, 14]), we observed that most of the complexity in these systems relates to the management of various forms of asynchronously-updated state, including sessions, protocols, and storage. Although quite complex, few of these systems involve intricate, uninterrupted sequences of computational steps. Hence, we suspected that datacenter infrastructure might be a good initial litmus test for our hypotheses about building distributed software.

In this paper, we report on our experiences building *BOOM Analytics,* an API-compliant reimplementation of the HDFS distributed file system and the Hadoop MapReduce engine. We named these two components *BOOM-FS* and *BOOM-MR*, respectively.[1] In writing BOOM Analytics, we preserved the Java API "skin" of HDFS and Hadoop, but replaced complex internal state with a set of relations, and replaced key system logic with code written in a declarative language.

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 missing significant distributed systems features like availability and scalability of master nodes. This allowed us to evaluate the difficulty of extending BOOM Analytics with complex features not found in the original codebase.

We implemented BOOM Analytics using the Overlog logic language, originally developed for Declarative Networking [24]. Overlog has been used with some success to prototype distributed system protocols, notably in a simple prototype of Paxos [34], a set of Byzantine Fault Tolerance variants [32], a suite of distributed file system consistency protocols [6], and a distributed hash table routing protocol implemented by our own group [24]. On the other hand, Overlog had not previously been used to implement a full-featured distributed system on the scale of Hadoop and HDFS. One goal of our work on BOOM Analytics was to evaluate the strengths and weaknesses of Overlog for system programming in the large, to inform the design of a new declarative framework for distributed programming.

1.2 Contributions

This paper describes our experience implementing and evolving BOOM Analytics, and running it on Amazon EC2. We document the effort required to develop BOOM Analytics in Overlog, and the way we were able to introduce significant extensions, including Paxos-supported replicated-master availability, and multi-master state-partitioned scalability. We describe the debugging tasks that arose when programming at this level of abstraction, and our tactics for metaprogramming Overlog to instrument our distributed system at runtime.

While the outcome of any software experience is bound in part to the specific programming tools used, there are hopefully more general lessons that can be extracted. To that end, we try to separate out (and in some cases critique) the specifics of Overlog as a declarative language, and the more general lessons of high-level data-centric programming. The more general data-centric aspect of the work is both positive and language-independent: many of the benefits we describe arise from exposing as much system state as possible via collection data types, and proceeding from that basis to write simple code to manage those collections.

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). These are noisy metrics, so we are most interested in numbers that transcend the noise terms: for example, order-of-magnitude reductions in code size. We also validate that the performance of BOOM Analytics is competitive with the original Hadoop codebase.

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 by replacing Hadoop's task scheduling logic with a declarative scheduling framework (Section 7). In each case, we discuss how the

[1] The BOOM Analytics software described in this paper can be found at `http://db.cs.berkeley.edu/eurosys-2010`.

data-centric approach influenced our design, and how the modifications involved interacted with earlier revisions. We compare the performance of BOOM Analytics with Hadoop in Section 8, and reflect on the experiment 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 [11] 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 [24], three-tier web services [38], natural language processing [13], modular robotics [5], video games [37], file system metadata analysis [15], and compiler analysis [20].

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 [37] — are algebraic or dataflow languages, used to describe the composition of operators that produce and consume sets or streams of data. Although arguably imperative, they are far closer to logic languages than to traditional imperative languages like Java or C, and are often amenable to set-oriented optimization techniques developed for declarative languages [37]. Declarative and dataflow languages can also share the same runtime, as demonstrated by recent integrations of MapReduce and SQL in Hive [35], DryadLINQ [39], 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 [28] and a transactional DHT called Scalaris with Paxos support [29]. Philosophically, Erlang revolves around concurrent *actors*, rather than data. Experience papers regarding Erlang can be found in the literature (e.g., [8]), and this paper can be seen as a complementary experience paper on building distributed systems in a data-centric fashion. A closer comparison of actor-oriented and data-centric design styles 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) [25]. By contrast, our approach is grounded in Datalog and its extensions.

Our use of metaprogrammed Overlog was heavily influenced by the Evita Raced Overlog metacompiler [10], and the security and typechecking features of Logic Blox' LB-Trust [26]. Some of our monitoring tools were inspired by Singh et al. [31], although our metaprogrammed implementation avoids the need to modify the language runtime as was done in that work.

```
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 RECURSIVE path(Start, End, NextHop, Cost) AS
(   SELECT From, To, To, Cost FROM link
    UNION
    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 all paths from links, along with an SQL translation.

2. Background

The Overlog language is sketched in a variety of papers. Originally presented as an event-driven language [24], it has evolved a semantics more carefully grounded in Datalog, the standard deductive query language from database theory [36]. Our Overlog is based on the description by Condie et al. [10]. 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:

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

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 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 all paths from links [23] is shown in Figure 1 (ignoring the Overlog-specific `@` notation), along with an SQL translation. Note how the SQL `WHERE` clause corresponds to the repeated use of the variable `To` 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. 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 `@`.

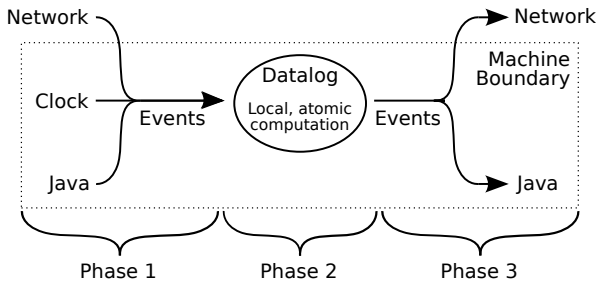

Figure 2. An Overlog timestep at a participating node: incoming events are applied to local state, the local Datalog program is run to fixpoint, and outgoing events are emitted.

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 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. The second phase interprets the local rules and tuples according to traditional Datalog semantics, executing the rules to a "fixpoint" in a traditional bottom-up fashion [36], 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 a static database, the first and third phases allow Overlog programs to mutate state over time.

2.1 JOL

The original Overlog implementation (*P2*) is aging and targeted at network protocols, so we developed a new Java-based Overlog runtime we call *JOL*. Like P2, JOL compiles Overlog programs into pipelined dataflow graphs of operators (similar to "elements" in the Click modular router [19]). JOL provides *metaprogramming* support akin to P2's Evita Raced extension [10]: each Overlog program is compiled into a representation that is captured in rows of tables. Program testing, optimization and rewriting can be written concisely as metaprograms in Overlog that manipulate those tables.

Because the Hadoop stack is implemented in Java, we anticipated the need for tight integration between Overlog and Java code. Hence, JOL supports Java-based extensibility in the model of Postgres [33]. 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 relations. We made significant use of each of these features in BOOM Analytics.

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 [14], 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 *chunks* and distributed across a set of *DataNodes*. By default, each chunk is 64MB and is replicated 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.

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. Following this lead, we implemented the simple high-bandwidth data path "by hand" in Java, concentrating our Overlog code on the trickier control-path logic. This allowed us to use a prototype version of JOL that focused on functionality more than performance. As we document in Section 8, this was sufficient to allow BOOM-FS to keep pace with HDFS in typical MapReduce workloads.

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.

[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.

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 together make up the primary key of each relation.

The NameNode must ensure that 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 [30] to write durable state changes to disk as an atomic transaction at the end of each fixpoint. Like P2, JOL allows durability to be specified on a per-table basis. So the relations in Table 1 were marked durable, whereas "scratch tables" that are used to compute responses to file system requests were transient — emptied at the end of each fixpoint.

Since a file system is naturally hierarchical, the "queries" needed to traverse it are recursive. While recursion in SQL is considered somewhat esoteric, it is a common pattern in Datalog and hence Overlog. 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. Note that when a *file* representing a directory is removed, all *fqpath* tuples that describe child paths of that directory are automatically removed (because they can no longer be derived from the updated contents of *file*).

Because path information is accessed frequently, we configured the *fqpath* relation to be cached after it is computed. Overlog will automatically update *fqpath* when *file* is changed, using standard relational view maintenance logic [36]. 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 altering 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. The data protocol was implemented in Java because it is simple and performance critical. We proceed to describe the three protocols in order.

```
// 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 deriving fully-qualified pathnames from the base file system metadata in BOOM-FS.

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 to the NameNode are handled in a serial fashion by JOL. While this simple approach has been sufficient for our experiments, we plan to explore more sophisticated concurrency control techniques in the future.

The heartbeat protocol follows a similar request/response pattern, but it is not driven by the arrival of network events. In order to trigger such events in a data-centric language, Overlog offers a *periodic* relation [24] that can be configured to produce new tuples at every tick of a wall-clock timer. DataNodes use the *periodic* relation to send heartbeat messages to NameNodes.

The NameNode can also send control messages to DataNodes. This occurs when a file system invariant is unmet and the NameNode requires the cooperation of the DataNode to restore the invariant. For example, the NameNode records the number of replicas of each chunk (as reported by heartbeat messages). If the number of replicas of a chunk drops below the configured replication factor (e.g., due to a DataNode failure), the NameNode sends a message to a DataNode that stores the chunk, asking it to send a copy of the chunk to another DataNode.

Finally, the data protocol is a straightforward mechanism for transferring the contents of a chunk between clients and DataNodes. This protocol is orchestrated by Overlog rules but implemented in Java. When an Overlog rule deduces

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.

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 in Overlog, and it was straightforward to add Java code to store chunks in UNIX files. Adding metadata durability took about a day. Adding the necessary Hadoop client APIs in Java took an additional week. As Table 2 shows, BOOM-FS contains an order of magnitude less code than HDFS. The DataNode implementation accounts for 414 lines of the Java in BOOM-FS; the remainder is devoted to system configuration, bootstrapping, and a client library. Adding support for accessing BOOM-FS via Hadoop's API required an additional 400 lines of Java.

In retrospect, the main benefit of our data-centric approach was to expose the simplicity of HDFS's core state, which consists of simple file system metadata and streams of messages in a few communication protocols. Having identified the relevant data and captured it in relations, the task of writing code to coordinate the data was relatively easy and could have been written fairly quickly in any language with good support for collection types.

Beyond this data-centric approach, the clearest benefit of Overlog's declarativity at this stage turned out to be the ability to (a) express paths as simple recursive queries over parent links, and (b) flexibly decide when to maintain materialized views (i.e., cached or precomputed results) of those paths separate from their specification.[3] Overlog's built-in support for persistence, messaging, and timers were also convenient, and enabled file system policy to be stated concisely.

When we began this work, we expected that using a declarative language would allow the natural specification and maintenance of file system invariants. We found that this was only partially true. For NameNode-local invariants (e.g., ensuring that the *fqpath* relation is consistent with the *file* relation), Overlog gave us confidence in the correctness of our system. However, Overlog was less useful for describing invariants that require the coordination of multiple nodes (e.g., ensuring that the replication factor of each chunk is satisfied). On reflection, this is because distributed Overlog rules induce asynchrony across nodes; hence, such rules must describe *protocols* to enforce distributed invariants, not the invariants themselves. Hence, the code we wrote to maintain the replication factor of each chunk had a low-level, "state machine"-like flavor. We return to this point in Section 9.2.

Although BOOM-FS replicates the basic architecture and functionality of HDFS, we did not attempt to achieve feature parity. HDFS features that BOOM-FS does not support include file access permissions, a web interface for status monitoring, and proactive rebalancing of chunks among DataNodes in a cluster. Like HDFS, the initial BOOM-FS prototype avoids distributed systems and parallelism challenges by implementing coordination with a single centralized NameNode. It 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.

4. The Availability Rev

Having achieved a fairly faithful implementation of HDFS, we were ready to explore whether data-centric programming would make it easy to add complex distributed functionality to an existing system. We chose what we considered a challenging 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 ([22] issue HADOOP-4539). We felt that a hot standby scheme would be more useful, and would more aggressively test our hypothesis that significant distributed system infrastructure could be implemented cleanly in a data-centric manner.

4.1 Paxos Implementation

Implementing hot standby replication is tricky, since replica state must remain consistent in the face of node failures and lost messages. One solution is to use a globally-consistent distributed log, which guarantees a total ordering over events affecting replicated state. Lamport's Paxos algorithm is the canonical mechanism for this feature [21].

We began by creating an Overlog implementation of basic Paxos, focusing on correctness and adhering as closely as possible to the initial specification. Lamport's description of Paxos is given in terms of "ballots" and "ledgers," which correspond to network messages and stable storage, respectively. The consensus algorithm is given as a collection of logical invariants which describe when agents cast ballots and commit writes to their ledgers. In Overlog, messages and disk writes are represented as insertions into tables with different persistence properties, while invariants are expressed as Overlog rules. Our first effort was clean and fairly simple: 22 Overlog rules in 53 lines of code, corresponding nearly line-for-line with the invariants from Lamport's original paper [21]. Since our entire implementation fit on a single screen, we were able to visually confirm its faithfulness to the original specification. To this point, working with a data-centric language was extremely gratifying, as we further describe in [4].

[3] In future, these decisions could be suggested or made automatic by an optimizer based on data and workloads.

Next, we needed to convert basic Paxos into a working primitive for a distributed log. This required adding the ability to efficiently pass a series of log entries ("Multi-Paxos"), a liveness module, and a catchup algorithm. While the first was for the most part a simple schema change, the latter two caused our implementation to swell to 50 rules in roughly 400 lines of code. Echoing the experience of Chandra et al. [9], these enhancements made our code considerably more difficult to check for correctness. The code also lost some of its pristine declarative character; 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 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 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. We present performance results in the technical report [2].

4.3 Discussion

Our Paxos implementation constituted roughly 400 lines of code and required six person-weeks of development time. Adding Paxos support to BOOM-FS took two person-days and required making mechanical changes to ten BOOM-FS rules (as described in Section 4.2). We suspect that the rule modifications required to add Paxos support could be performed as an automatic rewrite.

Lamport's original paper describes Paxos as a set of logical invariants. This specification naturally lent itself to a data-centric design in which "ballots," "ledgers," internal counters and vote-counting logic are represented uniformly as tables. However, as we note in a workshop paper [4], the principal benefit of our approach came directly from our use of a rule-based declarative language to encode Lamport's invariants. We found that we were able to capture the design patterns frequently encountered in consensus protocols (e.g., multicast, voting) via the composition of language constructs like aggregation, selection and join.

In our initial implementation of basic Paxos, we found that each rule covered a large portion of the state space, avoiding the case-by-case transitions that would need to be specified in a state machine-based implementation. However, choosing an invariant-based approach made it harder to adopt optimizations from the literature as the code evolved, in part because these optimizations were often described using state machines. We had to choose between translating the optimizations "up" to a higher-level while preserving their intent, or directly "encoding" the state machine into logic, resulting in a lower-level implementation. In the end, we adopted both approaches, giving sections of the code a hybrid feel.

5. The Scalability Rev

HDFS NameNodes manage large amounts of file system metadata, which are kept in memory to ensure good performance. The original GFS paper acknowledged that this could cause significant memory pressure [14], 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*. Having exposed the system state in tables, this was straightforward: it involved adding a "partition" column to various tables to split them across nodes in a simple way. Once this was done, the code to query those partitions — regardless of language in which it is written — composes cleanly with our availability implementation: each NameNode-partition can be deployed either as 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 every 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 local information to determine the correct NameNode-partition.

We did not attempt to support atomic "rename" across partitions. This would involve the atomic transfer of state between independent Paxos groups. We believe this would be relatively straightforward to implement — we have previously built a two-phase commit protocol in Overlog [4] — but we 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 eight hours of developer time to implement NameNode partitioning; two of these hours were spent adding partitioning and broadcast support to the BOOM-FS client library. This was a clear win for the data-centric approach, independent of any declarative features of Overlog.

Before attempting this work, we were unsure whether partitioning for scale-out would compose naturally with state replication for fault tolerance. Because scale-out in BOOM-FS amounted to little more than partitioning data collections, we found it quite easy to convince ourselves that

our scalability improvements integrated correctly with Paxos. Again, this was primarily due to the data-centric nature of our design. Using a declarative language led to a concise codebase that was easier to understand, but the essential benefits of our approach would likely have applied to a data-centric implementation in a traditional imperative language.

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. As Singh et al. observed, Overlog is in essence a stream query language, well-suited to writing distributed monitoring queries [31]. This offers a naturally introspective approach: simple Overlog queries can monitor complex protocols. Following that idea, we decided to develop a suite of debugging and monitoring tools for our own use in Overlog.

6.1 Invariants

One advantage of a logic-oriented language like Overlog is that system invariants can easily be written declaratively and enforced by the runtime. This includes "watchdog rules" that provide runtime checks of program behavior. For example, a simple watchdog rule can check that the number of messages sent by a protocol like Paxos matches the specification.

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. Our use of the *die* relation is similar to the *panic* relation described by Gupta et al. [16].

We made extensive use of these local-node invariants in our code and unit tests. Although these watchdog rules increase the size of a program, they improve both reliability and readability. In fact, had we been coding in Java rather than Overlog we would likely have put the same invariants in natural language comments, and "compiled them" into executable form via hand-written routines below the comments (with the attendant risk that the Java does not in fact achieve the semantics of the comment). We found that adding invariants of this form was especially useful given the nature of Overlog: the terse syntax means that 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 Overlog rules are causing the lion's share of the runtime.

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 [10], 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. We ran our regression tests through this tool, and immediately found both "dead code" rules in our programs, and code that we knew needed to be exercised by the tests but was as-yet uncovered.

6.3 Discussion

The invariant assertions described in Section 6.1 are expressed in 12 Overlog rules (60 lines of code). We added assertions incrementally over the lifetime of the project; while a bit harder to measure than our more focused efforts, we estimate this at no more than 8 person-hours in total. The monitoring rewrites described in Section 6.2 required 15 rules in 64 lines of Overlog. We also wrote a tool to present the

trace summary to the end user, which constituted 280 lines of Java. Because JOL already provided the metaprogramming features we needed, it took less than one developer day to implement these rewrites.

Capturing parser state in tables had several benefits. Because the program code itself is represented as data, introspection is a query over the metadata catalog, while automatic program rewrites are updates to the catalog tables. Setting up traces to report upon distributed executions was a simple matter of writing rules that query existing rules and insert new ones.

Using a declarative, rule-based language allowed us to express assertions in a "cross-cutting" fashion. A watchdog rule describes a query over system state that must never hold: such a rule is both a specification of an invariant and a check that enforces it. The assertion need not be closely coupled with the rules that modify the relevant state; instead, assertion rules may be written as a independent collection of concerns.

7. MapReduce Port

In contrast to our clean-slate strategy for developing BOOM-FS, we built BOOM-MR, our MapReduce implementation, by replacing Hadoop's core scheduling logic with Overlog. Our goal in building BOOM-MR was to explore embedding a data-centric rewrite of a non-trivial component into an existing procedural system. MapReduce scheduling policies are one issue that has been treated in recent literature (e.g., [40, 41]). 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 an input chunk from the distributed file system, runs a user-defined map function, and partitions output key/value pairs into hash buckets on the local disk. Reduce tasks are created for each hash bucket. Each reduce task fetches the corresponding hash buckets from all mappers, sorts locally by key, runs a user-defined 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 [11].

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.

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 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 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 data, 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.

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 LATE policy proposed by Zaharia et al. [41]. 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 Zaharia et al.'s paper via just three lines of pseudocode, but their implementation of the policy for vanilla Hadoop required adding or modifying over 800 lines of Java — an order of magnitude more than our Overlog implementation. 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 HDFS NameNode, while the remaining 100 nodes served as slaves for running the Hadoop TaskTrackers and HDFS DataNodes. Each TaskTracker was configured to support executing up to two map tasks and two reduce tasks simultaneously. The master node ran on a "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.

LATE focuses on how to improve job completion time by reducing the impact of "straggler" tasks. To simulate stragglers, 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). We ran each experiment five times, and report the average over all runs. Figure 4 shows the reduce task duration 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 durations, because adding artificial load had 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 was scheduled at the beginning of the job. This first wave of reduce tasks cannot finish until all map tasks have completed, which increased the duration of these tasks as indicated in the right portion of the graph. The second wave of 200 reduce tasks did not experience delay due to unfinished map work since it was scheduled after all map tasks had finished. These shorter task durations are reported in the left portion of the graph. Furthermore, stragglers had less 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 FCFS policy ported from Hadoop. This echoes the results of Zaharia et al. [41]

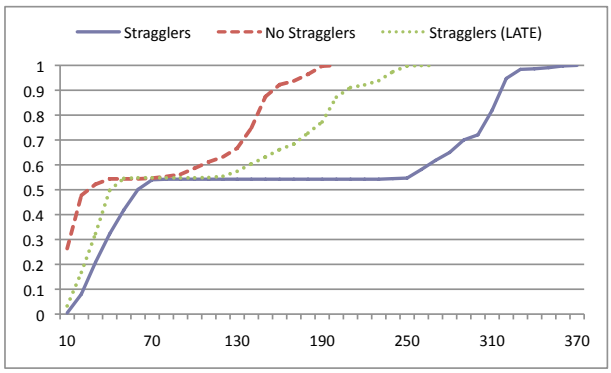

Figure 4. CDF of reduce task duration (secs), with and without stragglers.

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 leverage JOL's Java interfaces and 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.

We found that scheduling policies were a good fit for a declarative language like Overlog. In retrospect, this is because scheduling can be decomposed into two tasks: *monitoring* the state of a system and applying *policies* for how to react to changes to that state. Monitoring is well-handled by Overlog: we found that the statistics about TaskTracker state required by the LATE policy are naturally realized as aggregate functions, and JOL took care of automatically updating those statistics as new messages from TaskTrackers arrived. It is also unsurprisingly that a logic language should be well-suited to specifying policy. Overall, we found the BOOM-MR scheduler much simpler to extend and modify than the original Hadoop Java code, as demonstrated by our experience with LATE. Informally, the Overlog code in BOOM-MR seems about as complex as it should be: Hadoop's MapReduce task coordination logic is a simple and clean design,

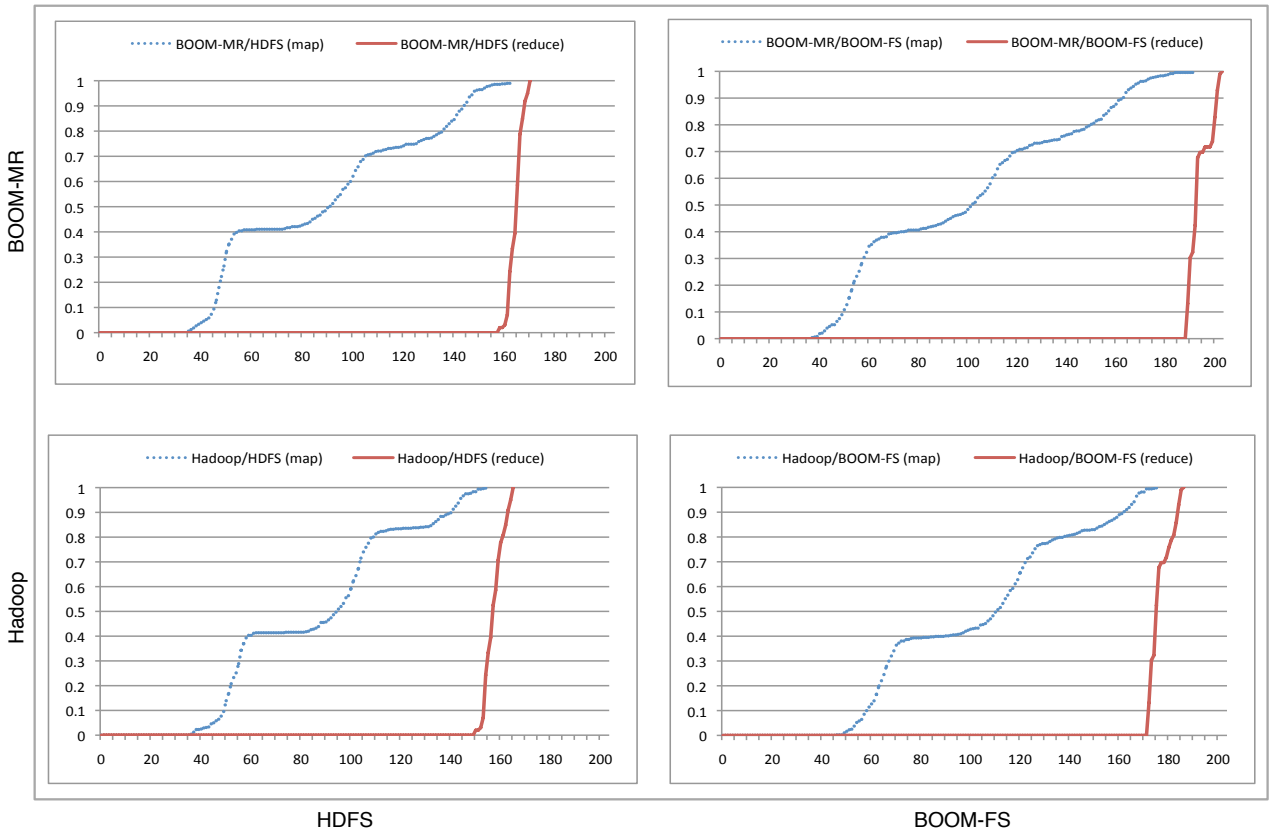

Figure 5. CDFs representing the elapsed time between job startup and task completion for both map and reduce tasks, for all combinations of Hadoop and BOOM-MR over HDFS and BOOM-FS. In each graph, the horizontal axis is elapsed time in seconds, and the vertical represents the percentage of tasks completed.

and the compactness of BOOM-MR reflects that simplicity appropriately.

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

9. Experience and Lessons

Our overall experience with BOOM Analytics has been quite positive. Building the system required only nine months of part-time work by four developers, including the time required to go well beyond the feature set of HDFS. Clearly, our experience is not universal: system infrastructure is only one class of distributed software, and analytics for "Big Data" is even more specialized. However, we feel that our experience sheds light on common patterns that occur in many distributed systems, including the coordination of multiple nodes toward common goals, replicated state for high-availability, state partitioning for scalability, and monitoring and invariant checking to improve manageability.

Anecdotally, we feel that much of our productivity came from using a data-centric design philosophy, which exposed the simplicity of tasks we undertook. When you examine the tables we defined to implement HDFS and MapReduce, it

seems natural that the system implementation on top should be fairly simple, regardless of the language used. Overlog imposed this data-centric discipline throughout the development process: no private state was registered "on the side" to achieve specific tasks. Beyond this discipline, we also benefited from a few key features of a declarative language, including built-in queries with support for recursion, flexible view materialization and maintenance, and high-level metaprogramming opportunities afforded by Overlog and implemented in JOL.

9.1 Everything Is Data

In BOOM Analytics, *everything* is data, represented as objects in collections. 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 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, all stored in a unified database representation (Section 6).

The uniformity of data-centric interfaces also allows *interposition* [18] 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 to 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.

In all, relatively little of this discussion seems specific to logic programming per se. Many modern languages and frameworks now support convenient high-level programming over collection types (e.g., list comprehensions in Python, query expressions in the LINQ framework [27], and algebraic dataflows like MapReduce). Programmers using those frameworks can enjoy many of the benefits we identified in our work by adhering carefully to data-centric "design patterns" when building distributed systems. However, this requires discipline in a programming language like Python, where data-centric design is possible but not necessarily idiomatic.

9.2 Developing in Overlog

Some of the benefits we describe above can be attributed to data-centric design, while others relate to the high-level declarative nature of Overlog. However, Overlog is by no means a perfect language for distributed programming, and it caused us various frustrations. Many of the bugs we encountered were due to ambiguities in the language semantics, particularly with regard to state update and aggregate functions. This is partly due to Overlog's heritage: traditional Datalog does not support updates, and Overlog's support for updates has never had a formally-specified semantics. We have recently been working to address this with new theoretical foundations [3].

In retrospect, we made very conservative use of Overlog's support for distributed queries: we used the `@` syntax as a convenient shorthand for unidirectional messaging, but we did not utilize arbitrary distributed queries (i.e., rules with two or more distinct location specifiers in their body terms). In particular, we were unsure of the semantics of such queries in the event of node failures and network partitions.

Instead, we implemented protocols for distributed messaging explicitly, depending on the requirements at hand (e.g., Paxos consensus, communication between NameNode and DataNodes). As we observed in Section 3.3, this made the enforcement of distributed invariants somewhat "lower-level" than our specifications of local-node invariants. By examining the coding patterns found in our hand-written messaging protocols, we hope to develop new higher-level abstractions for specifying and enforcing distributed invariants.

During our Paxos implementation, we needed to translate state machine descriptions 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. Now that we have achieved a functional Paxos implementation, we hope to revisit this topic with an eye toward rethinking the *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.

Finally, Overlog's syntax allows programs to be written concisely, but it can be difficult and time-consuming to read. Some of the blame can be attributed to Datalog's specification of joins via repetition of variable names. We are experimenting with an alternative syntax based on SQL's named-field approach.

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

Our experience developing BOOM Analytics in Overlog resulted in a number of observations that are useful on both long and short timescales. Some of these may be specific to our BOOM agenda of rethinking programming frameworks for distributed systems; a number of them are more portable lessons about distributed system design that apply across programming frameworks.

At a high level, the effort convinced us that a declarative language like Overlog is practical and beneficial for implementing substantial systems infrastructure, not just the isolated protocols tackled in prior work. Though our metrics were necessarily rough (code size, programmer-hours), we were convinced by the order-of-magnitude improvements in programmer productivity, and more importantly by our ability to quickly extend our implementation with substantial new distributed features. Performance remains one of our concerns, but not an overriding one. One simple lesson of our experience is that modern hardware enables "real systems" to be implemented in very high-level languages. We should use that luxury to implement systems in a manner that is simpler to design, debug, secure and extend — especially for tricky and mission-critical software like distributed services.

We have tried to separate the benefits of data-centric system design from our use of a high-level declarative language. Our experience suggests that data-centric programming can be useful even when combined with a traditional programming language, particularly if that language supports set-oriented data processing primitives (e.g., LINQ, list comprehensions). Since traditional languages do not necessarily encourage data-centric programming, the development of libraries and tools to support this design style is a promising direction for future work.

Finally, our experience highlighted problems with Overlog that emphasize some new research challenges; we mention two here briefly. First and most urgent is the need to codify the semantics of asynchronous computations and updateable state in a declarative language. We have recently made some progress on defining a semantic foundation for this [3], but it remains an open problem to surface these semantics to programmers in an intuitive fashion. A second key challenge is to clarify the implementation of invariants, both local and global. In an ideal declarative language, the specification of an invariant should entail its automatic implementation. In our experience with Overlog this was hampered both by the need to explicitly write protocols to test global invariants, and the multitude of possible mechanisms for enforcing invariants, be they local or global. A better understanding of the design space for invariant detection and enforcement would be of substantial use in building distributed systems, which are often defined by such invariants.

Acknowledgments

We would like to thank Peter Bodík, Armando Fox, Haryadi S. Gunawi, the anonymous reviewers, and our shepherd Jim Larus for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant Nos. 0713661, 0722077 and 0803690, the Air Force Office of Scientific Research under Grant No. FA95500810352, the Natural Sciences and Engineering Research Council of Canada, and gifts from IBM, Microsoft, and Yahoo!.

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. Dedalus: Datalog in time and space. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, Dec 2009.

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

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

[6] N. Belaramani et al. PADS: A policy architecture for data replication systems. In *NSDI*, 2009.

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

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

[9] T. D. Chandra et al. Paxos made live: an engineering perspective. In *PODC*, 2007.

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

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

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

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

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

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

[16] A. Gupta et al. Constraint checking with partial information. In *PODS*, 1994.

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

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

[19] E. Kohler et al. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

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

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

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

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

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

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

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

[27] F. Marguerie et al. *LINQ In Action*. Manning Publications Co., 2008.

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

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

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

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

[32] A. Singh et al. BFT protocols under fire. In *NSDI*, 2008.

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

[34] B. Szekely and E. Torres, Dec. 2005. `http://www.klinewoods.com/papers/p2paxos.pdf`.

[35] A. Thusoo et al. Hive - a warehousing solution over a Map-Reduce framework. In *VLDB*, 2009.

[36] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Company, 1990.

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

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

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

[40] M. Zaharia et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

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