Data-centric Programming for Distributed Systems
Chp2&3.2
by Peter Alvaro, 2015

presenter: Irene (Ying) Yu
2016/11/16
Outline

● Disorderly programming
● Overview for overlog
● Implementation in protocols (two-phase commit)
● Large-scale storage system (BOOM-FS)
● Revision for the implementation
● CALM Theroem
● Future work
Disorderly programming

- Hypothesis:
  - challenges of programming distributed systems arise from the mismatch between the sequential model of computation in which programs are specified as an ordered list of operations to perform

- What is disorderly programming
  - extends the declarative programming paradigm with a minimal set of ordering constructs
Why distributed programming is hard

The challenges of distributed programming systems

- concurrency
- asynchrony
- performance variability
- partial failure

asynchrony: uncertainty about the ordering and the timing

partial failure: some of computing components may fail to run, while others keep running without an outcome
Motivation

Problem

- All programmers must learn to be distributed programmers.
- Few tools exist to assist application programmers

- make distributed systems easier to program and reason about
- transform the difficult problem of distributed programming into problem of data-parallel querying
- design a new class of “disorderly” programming languages
  - concise expression of common distributed systems patterns
  - capture uncertainty in their semantics
Disorderly programming language

➢ encourages programmers to underspecify order (try to relax the dependence for order.)
➢ make it easy (and natural) to express safe and scalable computations
➢ extend the declarative programming paradigm with a minimal set of ordering constructs.
Background-Overlog

1. recursive query language extended from Datalog

2. combine data-centric design with declarative programming

head(A, C) :- clause1(A, B), clause2(B, C);
recv_msg(@A, Payload) :-
send_msg(@B, Payload), peers(@B, A);

least_msg(min<SeqNum>) :- queued_msgs(SeqNum, _);
next_msg(Payload) :- queued_msgs(SeqNum, Payload), least_msg(SeqNum);

SELECT payload FROM queued_msgs
WHERE seqnum = (SELECT min(seqnum) FROM queued_msgs);
Features

add notation to specify the data location

provide some SQL like extensions such as primary keys and aggregation.

define a model for processing and generate changes to tables.
Implementation-Consensus protocols

Difficulty: high-level → low-level

- increase program size
- increase complexity

2PC(two-phase commit)

Paxos

specified in the literature in a high level:

messages, invariants, and state machine transitions.
2PC implementation
2PC implementation
Two-phase commit

```
/* Count number of peers */
peer_cnt(Coordinator, count<Peer>) :-
    peers(Coordinator, Peer);

/* Count number of "yes" votes */
yes_cnt(Coordinator, TxnId, count<Peer>) :-
    vote(Coordinator, TxnId, Peer, Vote),
    Vote == "yes";

/* Prepare => Commit if unanimous */
transaction(Coordinator, TxnId, "commit") :-
    peer_cnt(Coordinator, NumPeers),
    yes_cnt(Coordinator, TxnId, NumYes),
    transaction(Coordinator, TxnId, State),
    NumPeers == NumYes, State == "prepare";

/* Prepare => Abort if any "no" votes */
transaction(Coordinator, TxnId, "abort") :-
    vote(Coordinator, TxnId, _, Vote),
    transaction(Coordinator, TxnId, State),
    Vote == "no", State == "prepare";

/* All peers know transaction state */
transaction(@Peer, TxnId, State) :-
    peers(@Coordinator, Peer),
    transaction(@Coordinator, TxnId, State);
```

"commit" or "abort"

NOT attempt to make progress in the face of node failures.

High level constructs (idioms):
- multicast(join)
- sequence
Timer

2 details for the impl:

- timeouts
- persistence

Coordinator will choose to abort if response of peers takes too long.

```c
/* Declare a timer that fires once per second */
timer(ticker, 1000ms);

/* Start counter when TxnId is in "prepare" state */
tick(Coordinator, TxnId, Count) :-
  transaction(Coordinator, TxnId, State),
  State == "prepare",
  Count := 0;

/* Increment counter every second */
tick(Coordinator, TxnId, NewCount) :-
ticker(),
tick(Coordinator, TxnId, Count),
NewCount := Count + 1;

/* If not committed after 10 sec, abort TxnId */
transaction(Coordinator, TxnId, "abort") :-
tick(Coordinator, TxnId, Count),
transaction(Coordinator, TxnId, State),
Count > 10, State == "prepare";
```
BOOM-FS (Berkeley Order of Magnitude)

An API-compliant reimplemention of the HDFS (Hadoop distributed file system) using overlog in internals

- high availability master nodes (via an implementation of MultiPaxos in Overlog)
- scale-out of master nodes to multiple machines (via simple data partitioning)
- unique reflection-based monitoring and debugging facilities (via metaprogramming in Overlog)
Working of HDFS

HDFS

metadata ops

heartbeat

data operations

NameNode

Data Nodes
relations in file system

- represent the file system metadata as a collection of relations.
- query over this schema

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Relevant attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Files</td>
<td>fileid, parentfileid, name, isDir</td>
</tr>
<tr>
<td>fqpath</td>
<td>Fully-qualified pathnames</td>
<td>path, fileid</td>
</tr>
<tr>
<td>fchunk</td>
<td>Chunks per file</td>
<td>chunkid, fileid</td>
</tr>
<tr>
<td>datanode</td>
<td>DataNode heartbeats</td>
<td>nodeAddr, lastHeartbeatTime</td>
</tr>
<tr>
<td>hb_chunk</td>
<td>Chunk heartbeats</td>
<td>nodeAddr, chunkid, length</td>
</tr>
</tbody>
</table>

Table 2.2: BOOM-FS relations defining file system metadata.
eg. derive fqpath from file

```prolog
// 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;
```

Listing 2.6: Example Overlog for computing fully-qualified pathnames from the base file system metadata in BOOM-FS.

- a recursive query language like Overlog was a natural fit for expressing file system policy.
protocols in BOOM-FS

➢ metadata protocol

clients and NameNodes use it to exchange file metadata

➢ heartbeat protocol

DataNodes use it to notify the NameNode

➢ data protocol

clients and DataNodes use it to exchange chunks.
metadata protocol

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

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

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

namenode rules

- specify the result tuple should be stored at client
- handle errors and return failure message

Listing 2.7 return the set of DataNodes that hold a given chunk in BOOM-FS
Evaluation

<table>
<thead>
<tr>
<th>System</th>
<th>Lines of Java</th>
<th>Lines of Overlog</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDFS</td>
<td>21,700</td>
<td>0</td>
</tr>
<tr>
<td>BOOM-FS</td>
<td>1,431</td>
<td>469</td>
</tr>
</tbody>
</table>

Table 2.3: Code size of two file system implementations

- similar performance, scaling and failure-handling properties to those of HDFS
- can tolerate DataNode failures but has a single point of failure and scalability bottleneck at the NameNode.
- consists of simple message handling and management of the hierarchical file system namespace.
Validation for the performance

Figure 2.2: CDFs representing the elapsed time between job startup and task completion for both map and reduce tasks.

Conclusion: BOOM-FS performance is slightly worse than HDFS, but remains very competitive.
Revision

- Availability
- Scalability
- Monitoring
Availability Rev

Goal: retrofitting BOOM-FS with high availability failover

- Implemented using a globally-consistent distributed log represented using Paxos
  - Guarantees a consistently ordered sequence of events over state replicas
  - Supports replication of distributed filesystem metadata

- All state-altering events are represented in BOOM_FS as Paxos Decrees
  - Passed into Paxos as a single Overlog rule
  - Stores tentative actions in intermediate table (actions not yet complete)

- Actions are considered complete when they are visible in a table join with the local Paxos log
  - Local Paxos log contains completed actions
  - Maintains globally accepted ordering of actions
Availability Rev - Validation

<table>
<thead>
<tr>
<th>Number of NameNodes</th>
<th>Failure Condition</th>
<th>Avg. Completion Time (secs)</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>None</td>
<td>101.89</td>
<td>12.12</td>
</tr>
<tr>
<td>3</td>
<td>None</td>
<td>102.70</td>
<td>9.53</td>
</tr>
<tr>
<td>3</td>
<td>Backup</td>
<td>100.10</td>
<td>9.94</td>
</tr>
<tr>
<td>3</td>
<td>Primary</td>
<td>148.47</td>
<td>13.94</td>
</tr>
</tbody>
</table>

Table 2.4: Job completion times with a single NameNode, 3 Paxos-enabled NameNodes, backup NameNode failure, and primary NameNode failure

- **Criteria**
  - Paxos operation according to specs at fine grained level
  - Evaluate high availability by triggering master failures

- What is the impact of the consensus protocol on system performance?
- What is the effect of failures on completion time?
- how the implementation will perform when the master fails?
Scalability Rev

NameNode is scalable across multiple NameNode-partitions.

- adding a “partition” column to the Overlog tables containing NameNode state
- use a simple strategy based on the hash of the fully qualified pathname of each file
- modified the client library
- No support atomic “move” or “rename” across partitions
Monitoring and Debugging Rev

Singh et al. idea: Overlog queries can monitor complex protocols

- convert distributed overlog rules into global invariants
- added a relation called die to JOL
  - java event listener is triggered when tuples are inserted into die relation
  - body: overlog rule with invariant check
  - head: die relation

increase the size of a program VS improve readability and reliability.
Monitoring via Metaprogramming

- replicate the body of each rule in an Overlog program
- send its output to a log table

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

eg. the Paxos rule that tests whether a particular round of voting has reached quorum:

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

Consistency And Logical Monotonicity (CALM).

- logically monotonic distributed code is *eventually consistent* without any need for coordination protocols (distributed locks, two-phase commit, paxos, etc.)
- eventual consistency can be *guaranteed* in any program by protecting non-monotonic statements (“points of order”) with coordination protocols.
Monotonic logic:

As input set grows, output set does not shrink

“Mistake-free”

Order independent

Expressive but sometimes awkward

e.g., selection, projection and join

Monotonic programs are therefore easy to distribute and can tolerate message reordering and delays

Non-Monotonic Logic

New inputs might invalidate previous outputs

Requires coordination

Order sensitive

e.g., aggregation, negation
Minimize Coordination

When must we coordinate?

- In cases where an analysis cannot guarantee monotonicity of a whole program

how should we do to coordinate?

- Dedalus, Bloom
Use CALM principle

monotonicity: develop checks for distributed consistency (no coordination)

- non-monotonic symbols are not contained (NOT, IN)
- semantics of predicates eg. MIN(x)<100

non-monotonicity: provide a conservative assessment (need coordination)

- flag all non-monotonic predicates in a program
- add coordination logic at its points of order.
- visualize the Points of Order in a dependency graph
Conclusion

- Using tables as a uniform data representation simplified the problem of state management.
- Natural to express these systems and protocols with high-level declarative queries, describing continuous transformations over that state.
- The uniformity of data-centric interfaces also enabled interposition of components in a natural manner.
- Time-stepped dataflow execution model is simpler than traditional notions of concurrent programming.
Weaknesses of overlog

● ambiguous temporal semantics:
  ○ not easy to express the info accumulation and state change using implication
● semantics does not model asyn communication.
  ○ unable to characterize uncertainty about when or whether the conclusions of such an implication will hold.
Future work

- disorderly debugging of large-scale data management systems
- unify the analysis techniques developed in this thesis
- explore hybrid approaches that use data lineage to communicate details about consistency anomalies back to programmers

Thanks!