

MapReduce on a Chord Distributed Hash Table

Andrew Rosen

Brendan Benshoof
Anu G. Bourgeois

Robert W. Harrison

1 Introduction

- Background
- Goals

2 ChordReduce

- System Architecture
- Chord
- CFS
- MapReduce Module

3 Experiments

- Experiment Details
- Results
- Conclusions

Background

- Google's MapReduce [1] paradigm is integral to data processing.
- Popular platforms for MapReduce, such as Hadoop [2], are designed to be used in datacenters with a degree of centralization.
- Until recently, analysis and optimization of MapReduce has largely remained constrained to that context.

Goals

- We wanted build a more abstract system for MapReduce.
- We remove core assumptions [3]:
 - The system is centralized.
 - Processing occurs in a static network.
- The resulting system must be:
 - Fault tolerant.
 - Scalable.
 - Completely decentralized.

Features of ChordReduce

ChordReduce is a decentralized framework for distributed computing:

- Scalable.
- Load-Balancing.
- Decentralized:
 - No centralized node is needed to maintain metadata.
 - No central coordinator for tasks.
- Fault tolerant:
 - The loss of multiple nodes does not impact integrity.
 - The network can withstand numerous simultaneous faults.
 - Nodes in the network autonomously repair damage.

System Architecture

ChordReduce has three layers

- Chord [4], which handles routing and lookup.
- The Cooperative File System (CFS) [5], which handles storage and data replication.
- MapReduce.

Chord

Chord is a distributed hash table (DHT), where the nodes in the network are arranged in a ring overlay.

- Nodes and files are assigned a m -bit key.
- Chord gives a high probability $\log_2 N$ lookup time for any key.
- Nodes know their predecessors and successors in the ring.
- Nodes also maintain a table of m shortcuts, called fingers.
- Nodes are responsible for files with keys between their predecessor's and theirs.

A Chord Network

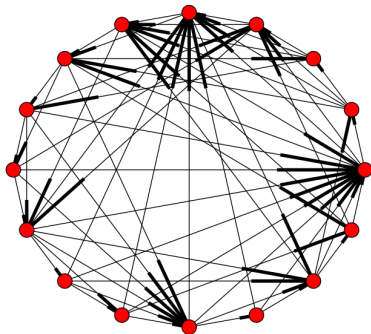


Figure: A Chord ring 16 nodes where $m = 4$.

CFS

The Cooperative File System runs on top of Chord.

- Files are split up, each block given a key based on their contents.
- Each block is stored according to their key.
- The hashing process guarantees that the keys are distributed near evenly among nodes.
- A keyfile is created and stored where the whole file would have been found.
- To retrieve a file, the node gets the keyfile and sends a request for each block listed in the keyfile.

Fault Tolerance

- Each node maintains a list of its s closest successors.
- Nodes back up data they're responsible for to their successors.
- When a node's predecessor fails, the node can immediately take over.
- The network will only lose data if $s + 1$ successive nodes fail simultaneously.
- The chances of this are r^{s+1} , where r is the failure rate.

Starting a MapReduce Job

- Jobs can be started at an arbitrary node, denoted the *stager*.
- The stager retrieves the keyfile and sends a map task for each key.
 - This process can be streamlined recursively by bundling keys and sending them to the best finger.
 - The resulting flow of data resembles a tree [6].
- Once the stager has sent a map to every node, its job is done.

Data Flow

- Results can be sent back via the overlay, or by initiating a direct connection.
- If a node receives multiple reduce results, they reduced into one before being sent along.

Fault Tolerance of Map Jobs

- Each node backups their map tasks; removes it when the task is processed.
- If the immediate successor detects the node's failure, it takes over the task.
- If a node detects a new predecessor responsible for a key and map task pair in it's queue, it sends it to the predecessor.
- This allows node to further distribute the work during execution.

Fault Tolerance of Reduces

- Individual reduces are backed up in a similar manner; if the original holder of the reduce fails before the reduce is sent, his successor sends his backup.
- Results are sent back to a key, rather than to a specific node.
- This ensures that if node receiving the data fails, his successor will take over.






Experiments

Stuff about Chord

Summary of Results

Stuff about Chord

Questions?

-  J. Dean and S. Ghemawat, “Mapreduce: Simplified Data Processing on Large Clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
-  “Hadoop,” <http://hadoop.apache.org/>.
-  “Virtual hadoop,” <http://wiki.apache.org/hadoop/Virtual>
-  I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications,” *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, August 2001. [Online]. Available: <http://doi.acm.org/10.1145/964723.383071>
-  F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Wide-Area Cooperative Storage with CFS,” *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 202–215, 2001.



K. Lee, T. W. Choi, A. Ganguly, D. Wolinsky, P. Boykin, and R. Figueiredo, “Parallel Processing Framework on a P2P System Using Map and Reduce Primitives,” in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 1602–1609.