Summary for three papers

Bigtable: A Distributed Storage System for Structured Data

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Fi- nance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the sim- ple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we de- scribe the design and implementation of Bigtable.
We have described Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production use since April 2005, and we spent roughly seven person-years on design and implementa- tion before that date. As of August 2006, more than sixty projects are using Bigtable. Our users like the perfor- mance and high availability provided by the Bigtable im- plementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time.
Given the unusual interface to Bigtable, an interest- ing question is how difficult it has been for our users to adapt to using it. New users are sometimes uncertain of how to best use the Bigtable interface, particularly if they are accustomed to using relational databases that support general-purpose transactions. Nevertheless, the fact that many Google products successfully use Bigtable demon- strates that our design works well in practice.
We are in the process of implementing several addi- tional Bigtable features, such as support for secondary indices and infrastructure for building cross-data-center replicated Bigtables with multiple master replicas. We have also begun deploying Bigtable as a service to prod- uct groups, so that individual groups do not need to main- tain their own clusters. As our service clusters scale, we will need to deal with more resource-sharing issues within Bigtable itself [3, 5].
Finally, we have found that there are significant ad- vantages to building our own storage solution at Google. We have gotten a substantial amount of flexibility from designing our own data model for Bigtable. In addi- tion, our control over Bigtable's implementation, and the other Google infrastructure upon which Bigtable de- pends, means that we can remove bottlenecks and ineffi- ciencies as they arise.

MapReduce: Simplified Data Processing on Large Clusters

MapReduce is a programming model and an associ- ated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.
Programs written in this functional style are automati- cally parallelized and executed on a large cluster of com- modity machines. The run-time system takes care of the details of partitioning the input data, scheduling the pro- gram's execution across a set of machines, handling ma- chine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to eas- ily utilize the resources of a large distributed system.
Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many ter- abytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce pro- grams have been implemented and upwards of one thou- sand MapReduce jobs are executed on Google's clusters every day.
The MapReduce programming model has been success- fully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality opti- mization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce com- putations. For example, MapReduce is used for the gen- eration of data for Google's production web search ser- vice, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clus- ters of machines comprising thousands of machines. The implementation makes efficient use of these machine re- sources and therefore is suitable for use on many of the large computational problems encountered at Google.
We have learned several things from this work. First, restricting the programming model makes it easy to par- allelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization al- lows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle ma- chine failures and data loss.

The Google File System

We have designed and implemented the Google File Sys- tem, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.
While sharing many of the same goals as previous dis- tributed file systems, our design has been driven by obser- vations of our application workloads and technological envi- ronment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore rad- ically different design points.
The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our ser- vice as well as research and development efforts that require large data sets. The largest cluster to date provides hun- dreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.
In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.
The Google File System demonstrates the qualities es- sential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data pro- cessing tasks of a similar magnitude and cost consciousness.
We started by reexamining traditional file system assump- tions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically different points in the design space. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequen- tially), and both extend and relax the standard file system interface to improve the overall system.
Our system provides fault tolerance by constant moni- toring, replicating crucial data, and fast and automatic re- covery. Chunk replication allows us to tolerate chunkserver failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently re- pairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.
Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involve- ment in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to pri- mary replicas in data mutations. This makes possible a sim- ple, centralized master that does not become a bottleneck. We believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.
GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.